Euphrates: Algorithm-SoC Co-Design for Low-Power Mobile Continuous Vision

Paper by: Y. Zhu, A. Samajdar, M. Mattina, P. Whatmough, “Euphrates: algorithm-SoC co-design for low-power mobile continuous vision,” arXiv, Apr. 2018.

Presentation by: Peter Fitchen, Deepen Solanki, Matt Bernath

Abstract

- Continuous CV applications are increasingly reliant on CNNs.
    - Mobile/embedded devices often don’t have compute power and memory needed for SOTA CNN models.
    - Energy and compute efficiency of CNNs can be drastically improved with purpose-built hardware.
- Euphrates is proposed as an algorithm-based SoC architecture design to improve performance and energy consumption of continuous CV applications on mobile/embedded devices.
- Key idea: exploit inherent motion information to reduce dependency on expensive CNNs.

Introduction

- First, a new algorithm that leverages temporal pixel motion for continuous vision applications is proposed.
  - Lighter-weight than the current state-of-the-art, but with minimal performance loss.
- Then, the paper describes how an SoC architecture can be co-designed with the proposed algorithm to implement the model in hardware.
- Argued that continuous vision should be task-anonymous and minimize waking the main CPU. So, new hardware IP called a *motion controller* is introduced.

Introduction

- Euphrates is a proof-of-concept of their SoC architecture co-designed with their proposed algorithm.
  - Implemented/modelled by a few modifications to existing work.
  - Evaluated on the tasks of object tracking and object detection and benchmarked against SOTA models.
  - 66% energy savings with double the object detection rate.
  - 21% energy savings for object tracking.
  - Both benchmarks incur at best a 1% reduction in accuracy compared to SOTA models.

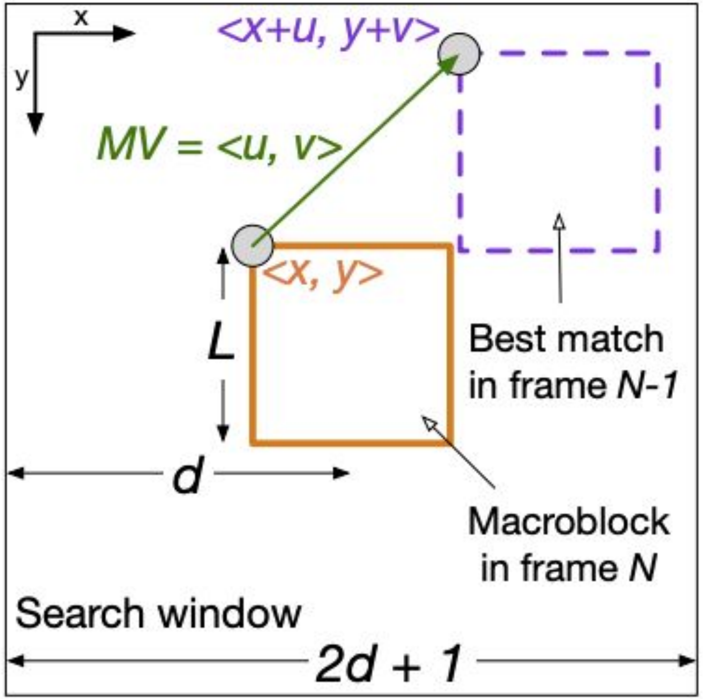
Background and Motivation

- Think of continuous CV pipeline in two parts:
  - Frontend that prepares pixel and metadata from camera sensor module input for the backend.
  - Backend that extracts semantic information for higher level decision making, often with a CNN.
- Frontend hardware includes:
  - Camera sensor module and an Image Signal Processor (ISP) unit (typically in the main SoC).
  - Raw pixels are converted to RGB/HSV frames that are passed to RAM.
  - Proposed algorithm also passes temporal pixel information (metadata) to RAM.
- Backend algorithms are typically expensive CNNs.
  - Often utilizes a DSP, GPU, FPU, and/or CPU, depending on what’s available.

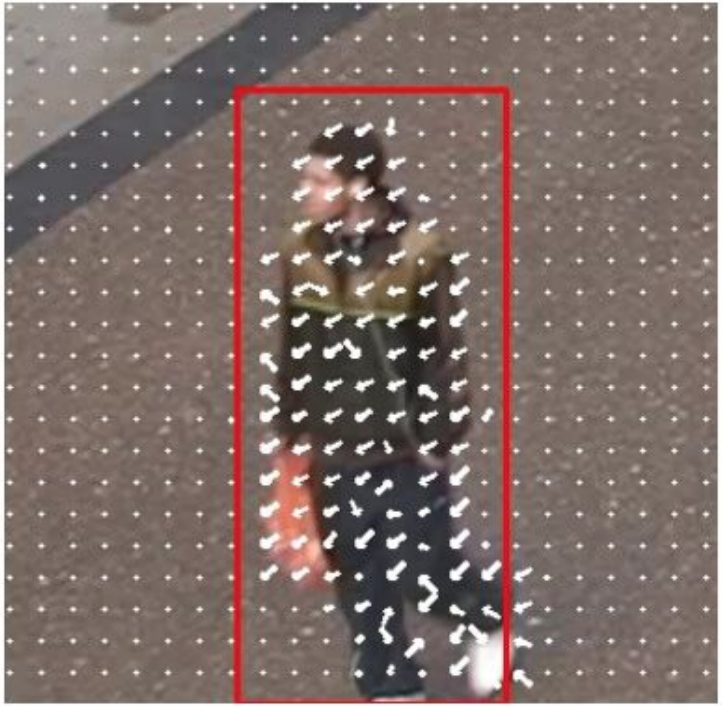
Motion Estimation Using Block-Matching (BM)

- Divides a frame into LxL macroblocks (MB) and search in the previous frame within a 2D search window (2d+1) pixels for the closest match using Sum of Absolute Differences.
- Various BM search strategies have different accuracy vs. compute efficiency tradeoffs. The most accurate method is to perform an exhaustive search, which requires $L^2(2d+1)^2$ operations for each MB.
- Another strategy is the Three Step Search (TSS), which logarithmically decreases d in steps to only search part of the window. It requires $L^2(1+8\log_2(d+1))$ operations, which is an 8/9 reduction for d=7.
- The eventual output of BM is a Motion Vector (MV) for each MB. The MV (<u, v>) describes how each MB moved from the previous frame to the current frame. MVs are efficient to store since they only require $\log_2(2d+1)$ bits (rounded up), which equates to 1 byte for d=7.

Motion Estimation Using Block-Matching (BM)



(a) Block-matching.



(b) Motion vectors.

Motion-Based Continuous Vision Algorithm

- Key idea is that pixel changes between frames directly encodes motion, and this temporal information can be used to simplify the motion extrapolation portion of continuous CV tasks.
- Two important aspects of the proposed algorithm are ***how*** and ***when*** to extrapolate from temporal information.
- Frames are categorized in two ways: (1) Inference Frames (I-frames) and (2) Extrapolation Frames (E-frames).
    - I-frames are those that are passed to expensive CNN algorithms and the like.
    - E-frames are those that extrapolate ROIs from the previous frame (could be an I or E frame).

Motion-Based Continuous Vision Algorithm

- ***How*** to extrapolate:
  - Calculate the average motion vector for each ROI (each pixel inherits the motion vector from the MB it is part of).
  - But, this introduces MV noise and doesn’t consider object deformation!
- Handling MV noise:
  - MV noise is modelled by attaching a confidence value to each MV, and this confidence value is heavily correlated with SAD values. A higher SAD value means lower confidence.
  - The SAD values are normalized w.r.t. their maximum value and set to fall between 0 and 1. The confidence for an ROI is then the average of these normalized values.
  - The MV for an ROI can then be filtered by a weighted average with the MV from the previous frame. The weighting is determined by the degree of confidence for the current MV. If it is above a threshold, it is the weight itself, otherwise the weight is set to a default value such as 0.5.
  - This final MV can then be used to translate the ROI from the previous frame to the current frame.
  - However, this still doesn’t model ROI deformation!

Motion-Based Continuous Vision Algorithm

- ***How*** to extrapolate:
- Handling Object Deformation:
  - Each ROI is actually split into sub-ROIs that each get their own MV and are updated accordingly.
  - This allows each sub-ROI to move in separate directions (consider a runner’s arms and legs, for example).
  - The final ROI is then the smallest bounding box that still contains each sub-ROI.

Motion-Based Continuous Vision Algorithm

How to extrapolate:

μ - average motion vector for an ROI

N - number of pixels in ROI

v_i - motion vector for each pixel (inherits from MB)

α - confidence value for each MB

SAD - Sum of Absolute Differences for each MB

L - MB dimension

β - weight for averaging

R - ROI

MV - Motion Vector for each ROI

$$\mu = \sum_i^N \vec{v_i} \,/\, N$$

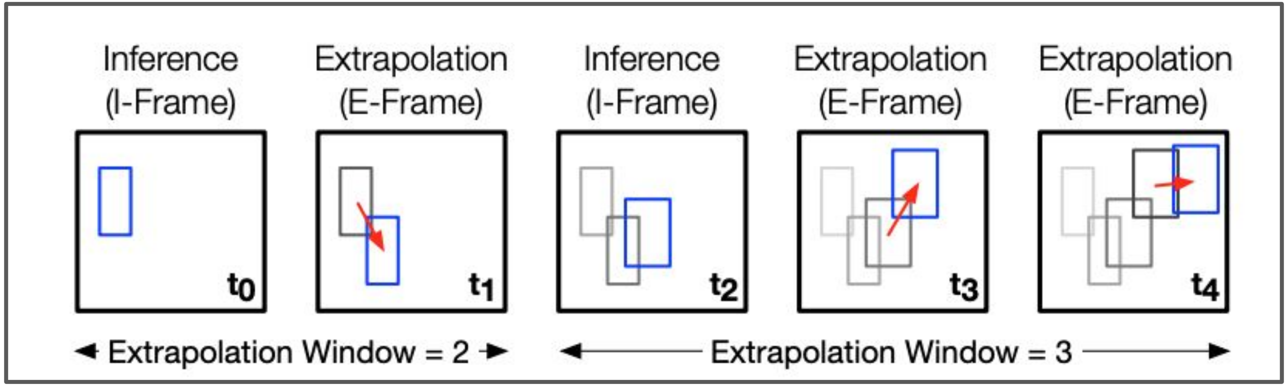
$$\alpha_F^i = 1 - \frac{SAD_F^i}{255 \times L^2}$$

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

$$R_F = R_{F-1} + MV_F$$

Motion-Based Continuous Vision Algorithm

- Mixing I and E frames can drastically improve compute efficiency.
  - Only use expensive CNNs for inference *some* of the time.



- However, using too many E-frames will degrade accuracy. Thus the question becomes how to strike the right balance (i.e. ***when*** to extrapolate).
- The notion of an Extrapolation Window (EW) is introduced.
  - The number of consecutive E-frames between I-frames + 1.
- A larger EW improves efficiency, but degrades accuracy.

Motion-Based Continuous Vision Algorithm

- ***When*** to extrapolate:
  - Euphrates provides two modes of setting the EW: (1) a constant mode and (2) an adaptive mode.
  - Constant mode is straightforward: EW is statically fixed at say 2 (in which case it is roughly twice as efficient/50% energy savings). But the accuracy tradeoff may not be ideal.
  - Adaptive mode will still calculate new ROI from extrapolation whenever an I-frame is processed.
  - If the results from inference and extrapolation are similar (above a threshold), the EW is incrementally increased. And if the results differ (or are below a similarity threshold), the EW is decreased.
  - Constant mode is advantageous when specific frame rate deadlines must be met, but without such restrictions adaptive mode will reduce energy consumption with minimal effect on accuracy.

Architecture Support

Principles

- Avoid unnecessary CPU usage - more architectural support in the vision pipeline
- The architectural support for the extrapolation functionality should be decoupled from CNN inference - flexible design, not restricted to limitations of CNN accelerators and their evolution, CNNs are expensive

Two architectural extensions

- Augment ISP to expose motion vectors to the backend - normally ISP does pre-processing and forgets
- Motion Controller to coordinate the backend

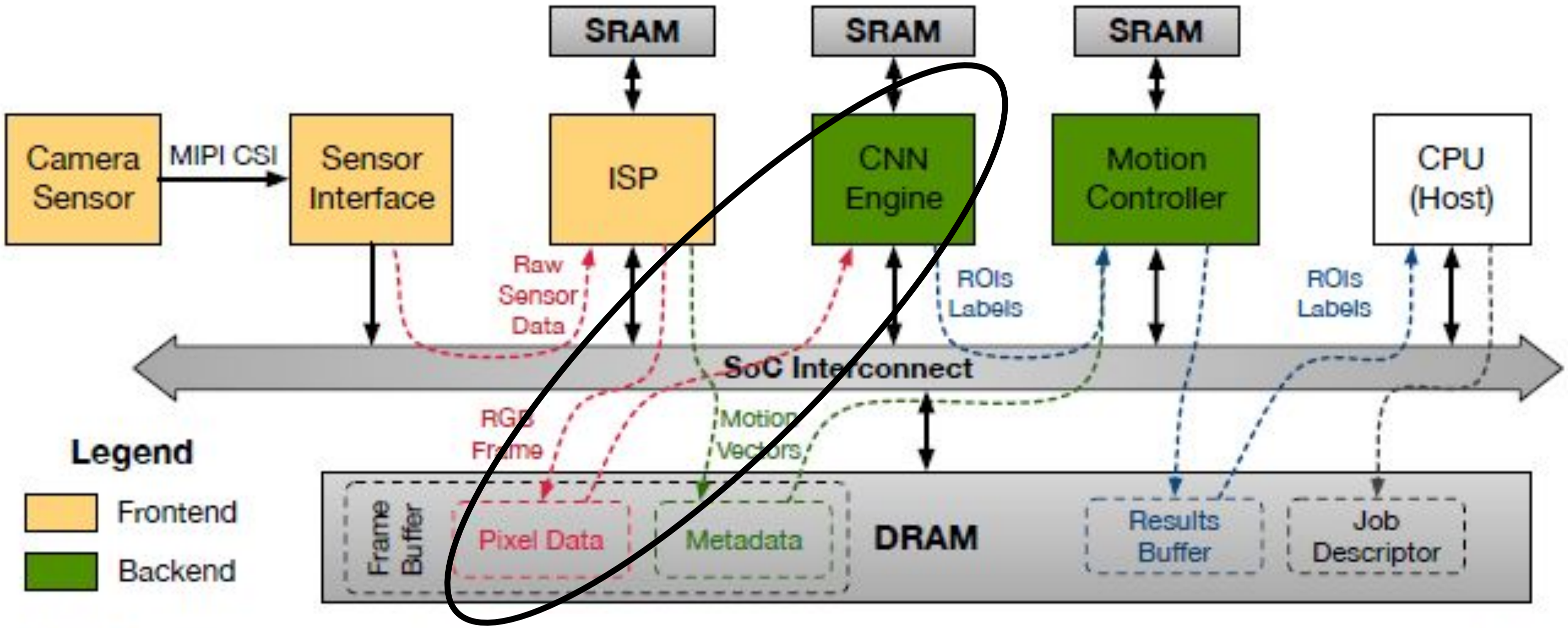


Fig. 5: Block diagram of the augmented continuous vision subsystem in a mobile SoC.

Architectural-Level Working

1. CPU configures components of the pipeline, initiates task
2. Camera captures images, feeds to ISP
3. ISP → pixel data + metadata, sent to DRAM buffer

---

4. CNN engine → inference pass → ROIs and possibly classification labels written to dedicated memory mapped registers in the Motion Controller
5. Motion Controller combines CNN data + MV data for E-Frames. Master slave communication, extrapolation window decision + choosing between I or E results

Motion Controller → Microcontroller-like, has an on-chip SRAM unlike Cortex-M, no fetch-decode, instead has a programmable sequencer,

Implementation and Experimental Setup

- An in-house hardware simulator was created with a methodology similar to Gemdroid SoC Simulator.
  - Simulator comprised of a functional model, performance model, and a power model to evaluate the continuous vision pipeline.
  - Functional model takes video streams to mimic real-time camera capture and implements the extrapolation algorithm in OpenCV.
  - Performance model captures the timing behaviors of pipeline components.
  - Power model is created by taking the timings of cross-IP activities, from which SoC events are tabulated and used for energy estimations.
    - Real device components were measured and used in the power estimation model.
- Software was evaluated using two popular mobile continuous vision scenarios:
  - Object detection scenario was evaluated using a custom dataset extracted from real video streams and was compared to Tiny YOLO.
  - Visual tracking scenario used CNN-based tracker called MDNet with two tracking benchmarks:
    - Object Tracking Benchmark (OTB) 100
    - VOT 2014

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

Evaluation: Object Detection

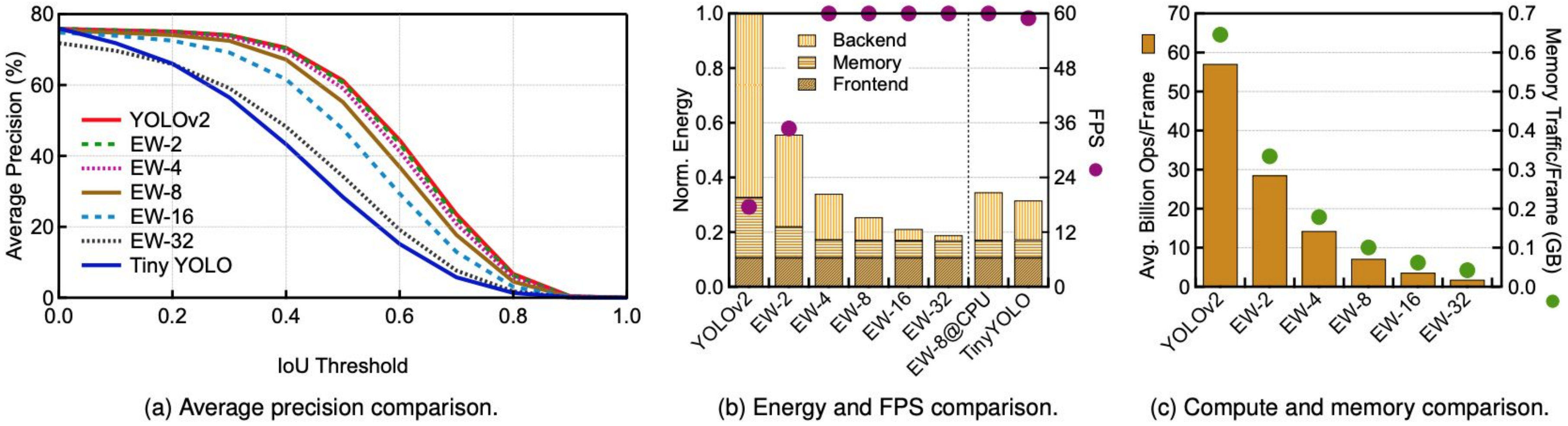
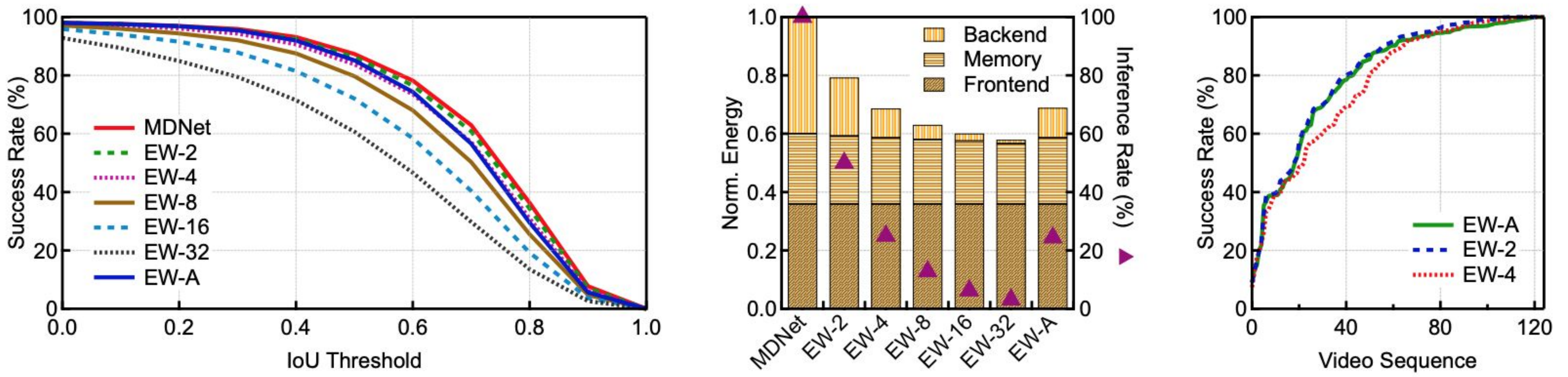


Fig. 9: Average precision, normalized energy consumption, and FPS comparisons between various object detection schemes. Energy is broken-down into three main components: backend (CNN engine and motion controller), main memory, and frontend (sensor and ISP).

Evaluation: Object Tracking



(a) Success rate comparison.

(b) Normalized energy consumption and inference rate comparison.

(c) Success rate for all 125 video sequences under an IoU ratio of 0.5.

Fig. 10: Accuracy loss and energy saving comparisons between baseline MDNet and various Euphrates configurations for OTB 100 and VOT 2014 datasets. Energy is dissected into three parts: backend (CNN engine and motion controller), main memory, and frontend (sensor and ISP).

Related Work

- CNN based models that use MVs as training input (Zhang, Chadha, TSN)
- YOLO → entire image in one go, regression problem.
- Fast YOLO → requires training a separate CNN, does not perform extrapolation
- Sliding window based methods
- Current research focus → design better accelerators, memory utilization, better tooling
- Euphrates → Motion extrapolation replaces inferencing.

Discussion

- Researchers note that Euphrates is least effective when dealing with scenes with fast moving and blurred objects.
    - They suspect it’s due to the limited scope of search window size in BM.
    - Unfortunately increasing the search window rapidly increases computational overhead.
    - A frame rate increase could improve performance.
- Researchers also suggest using sensor fusion algorithms with data from other sources, such as IMUs for more accurate motion estimation.

Questions ?