Study of an integrated pre-processing architecture for smart-imaging-systems, in the context of lowpower computer vision and embedded object detection

Luis Cubero Montealegre

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Nano-Electronique et Nano-Technologies**

Arrêté ministériel : 25 mai 2016

Présenté par

Luis Angel CUBERO MONTEALEGRE

Thèse dirigée par **Gilles SICARD,** Ingénieur de recherche, HDR, encadrée par **Arnaud PEIZERAT,** Ingénieur de recherche et **Dominique MORCHE,** Ingénieur de recherche, HDR.

préparée au sein du **CEA-LETI** dans **l' Ecole Doctorale d'Electronique, Automatisme et Traitement du signal (EEATS).**

Etude d'une architecture intégrée de prétraitement du signal pour les imageurs intelligents, dans le contexte de la vision embarquée pour la détection des objets.

Study of an integrated pre-processing architecture for smart-imaging-systems, in the context of low-power computer vision and embedded object detection

Thèse soutenue publiquement le **16/12/2021**,
devant le jury composé de :

Pr. Alice CAPLIER
GIPSA-Lab, Grenoble INP, Grenoble, France, Présidente
Dr. Ricardo CARMONA GALAN
Chercheur, Instituto de Microelectrónica de Sevilla, IMSE-CNM, Séville, Espagne, Rapporteur
Pr. Dominique GINHAC
Laboratoire ImViA – Université de Bourgogne, Dijon, France, Rapporteur
Dr. Gilles SICARD
Ingénieur de recherche HDR, CEA-LETI, Grenoble, France, Directeur de thèse
Dr. Arnaud PEIZERAT
Ingénieur de recherche, CEA-LETI, Grenoble, France, co-encadrant, invité
Dr. Dominique MORCHE
Ingénieur de recherche HDR, CEA-LETI, Grenoble, France, co-encadrant, invité

To my grandmothers.

Abstract

Embedded Computer vision, as many applications of artificial intelligence and edge computing, is subjected to hardware and power constraints. For instance, the object detection problem, consisting in finding different objects of specific classes (types) in an image, turns out to be quite complicated to embed near the image sensor as two complex tasks are required: multi-scale localization and multi-class classification (i.e. identifying bounding boxes that perfectly enclose each object, whatever its size, and labeling the type of the detected object). Today these tasks are often performed on general-purpose desktop machines. Nevertheless, attractive applications like autonomous-driving, augmented reality or video surveillance are urging the need for low-power, low-latency and compact low power devices.

The state of the art has approached this challenge by optimizing specific sections of the complete processing-pipeline for a comparable object detection performance. A typical example in the last decade corresponds to minimizing the computing precision, hence the power, to a minimal value. Diminishing the bit-depth or image size has then been studied while implementing pre-processing steps that increase robustness against the loss in bit and image resolution. An algorithm that does not require that kind of pre-processing stage to be programmable is obviously desirable in order to simplify its implementation (e.g. no memory access to learned weights). Another strategy has been to reduce power due to I/O communications amongst different chips or devices thanks to a more exhaustive integration of specialized circuitry and thanks to more efficient memory accesses and mathematical operations.

In that context of near-sensor computing, this work points towards a more energy efficient detection pipeline. We target several specific key aspects:

1. We try to assess if a dedicated-class-agnostic region proposal algorithm, based on pre-processed low-level features, could replace the typical sliding window approach for object localization in integrated smart imaging systems, allowing to target more efficiently objects in the image. Then, we propose a pipeline that takes into account near image sensor features extraction for Region Proposals with an embedded version of an algorithm called Edge-Boxes.

2. We try to assess an optimal type of pre-processing (based on an efficient architecture) that would allow extracting low level features (oriented gradients), and give the best trade-off between power consumption, hardware complexity and object detection performance. Specifically, while being this architecture fully compatible with region proposal algorithms beyond the sliding window.

3. Finally, we try to assess if non-standard, or neuromorphic, image acquisition techniques can be exploited in order to further increase the detection efficiency in real case scenarios.

Our methodology relies on behavioral simulations carried out thanks to a custom framework written in Python and C++ code. We propose a hierarchical model (and code architecture) of different image acquisition and processing techniques, and we study their performance through specific metrics related to runtime, memory usage, hardware complexity, I/O data-rate, localization performance and classification performance. We provide comparison with the state of the art and several benchmarks giving guidance to choose one or another architecture depending on the specific needs, and we conclude by stating which one would give, from our perspective, the best trade-offs.

Table of contents

List of figures

List of tables

Chapter 1. Introduction

Computer vision and artificial intelligence have opened a new range of applications. For instance, machines have become able to solve complex problems for object localization and classification, and also with high resolution images. That is, even if challenging characteristics such as high dynamic range, occlusion and significant affine/projective transformations are present. Moreover, most recent algorithms, based on deep convolutional neural networks (CNNs), have shown capacity to classify objects from an elevated amount of different classes.

Nevertheless, some of those new applications are simply incompatible with desktop-machine computations and/or servers: typical limitations in latency, autonomy, size, weight and financial cost constraint the computational unit to be compact and portable. Probably, one of the most common use-case scenarios are mobile phones. In addition, other examples where such limitations are present are drones, autonomous vehicles, video surveillance and wearable devices, among others. The aforementioned means that such smaller devices are logically less equipped, and thus it is common to say that they are under “hardware and power constraints”.

Regarding the last sentence, power consumption restrictions typically arise when devices are not connected to a “permanent” energy source, but they rather depend on a battery. Of course, for being smaller, the battery size has to be reduced too, and thus its lifetime (before re-charging) as well. Yet, computer vision tasks are often related to computationally intensive operations, and, for instance, CNNs are not an exception. In parallel, each computation implies energy, and the faster the system goes, the more power is drained from the battery. In the other hand, sending data out from the system to a remote server is possible, but the energy cost of this I/O bus is significant and often unsuitable if the data-rate is too high.

Last paragraphs sets motivations for one entire research field: one that is focused on computer vision and/or artificial intelligence on hardware/power constrained devices. The set of problems addressed commonly cannot be reduced to simply “take” the exact same algorithm and then “run it” on an embedded device. Typical reasons for that are that embedded devices have, for example, less memory available, a reduced instruction set (e.g. summation, subtraction, bit-shifting, etc…), and less energy compared to desktop computers or servers. Those reasons are why efforts are made in optimizing, simplifying and/or adapting algorithms and hardware, so even embedded systems can perform more complex computer vision tasks.

Continuing with last ideas, our work focuses on optimizing the power versus AI performance trade-off for object detection. Object detection is a known problem in artificial intelligence and computer vision: the problem is giving bounding-boxes for objects in a still image as the algorithm output, along with class labels for each object inside each bounding box. Moreover, as other groups do, our approach is to include more complex/adapted operations on the same integrated chip, so the I/O data-rate with remote servers and/or other chips is reduced. Furthermore, as other groups do as well, we optimize the algorithm/hardware to make it more energy-efficient. We take into account the development of newer microfabrication technologies allowing 3D-IC staking for justifying the addition of more complex circuitry.

More specifically, we tackle the last mentioned problem by optimizing the region-proposals stage and the low-level-features generation in the object detection pipeline. We explore the inclusion

of circuitry specialized on generating low-level hand-crafted features for region proposals. We also explore both frame-based (classic) and asynchronous/neuromorphic (non-classic) pixel topologies for achieving it. We believe that region proposals can replace other typical IC implementations that rely on the older sliding window approach, and for energy efficiency. That is, since the computational complexity of the sliding window makes it inefficient or even unsuitable for solving problems with a high-scale-range or with high-resolution images. Indeed, with the so-called sliding window, embedded object detection can be constrained to “small” images only, and with objects in positions and/or scales that are relatively “easy” to solve, or in other works, incompatible with real-case/outdoors scenarios.

This work has been made in a laboratory called CEA-LETI/L3I: a group specialized in CMOS image-vision-sensor development, and also in integrated, low-power image pre-processing electronic architectures. The laboratory is located in Grenoble, France, and it forms a part of a bigger entity called the Commissary of Atomic Energy and Renewable Energies (CEA). The CEA counts with different laboratories in a wide range of fields, from fundamental sciences to applied electronics.

Our manuscript is composed of eight chapters (counting the introduction and conclusion). Chapter two gives a brief view of the state of the art in CMOS image-sensors, smart-imagers and object detection algorithms. Chapter three presents our simulation and modeling methodology, which we use to derive conclusions for the electronic-architecture design. Chapter four explains which kind of object detection pipeline we propose to use for embedded applications. Chapter five presents the low-level features architecture that we propose to use for region-proposals-generation. Chapter six presents a series of benchmarks that show (at simulation level) that our pipeline is suitable for region-proposals-generation on constrained devices. Finally, chapter seven presents further models and simulation results regarding the implementation of non-classic, neuromorphic pixels for optimizing the region proposals generation as well.

Chapter 2. State of the art

Chapter 2 describes the main theoretical concepts for the general understanding of this thesis. We present a synthesis of our State of Art review, so the reader can find here a (non-exhaustive) list of related works. The first principal subject mentioned is "CMOS" Image sensors: we cover from photo-generation to digital-image-output. We do so for different light-sensing types, and for several pixel architectures. However, we do not go into details, as we only cite relevant ideas for later in this work. The second principal subject is what a "smart-image-sensor" is. We explain it by signaling key differences with classic architectures, and by citing illustrative examples in the state of the art. The final subject covered here is the "object-detection" problem: it comes from the fields of Artificial Intelligence, Machine-learning and Computer-vision. We describe its formal definition, and later we relate it with applications and implementations in the embedded-IC case-scenario.

2.1. CMOS Image sensors

The term « CMOS » is an acronym for « complementary metal oxide semiconductor » (El Gamal and Eltoukhy 2005). The name describes a specific type of electronics-related technology and physics knowledge, allowing creating devices known as CMOS transistors. CMOS transistors are useful to create simple or complex circuits, as switches, amplifiers and digital-processors, among many others. During advances in imaging techniques, CMOS technology became attractive for devices that generate a static image of a scene. In order to achieve that, transistors, combined with light-sensing devices (photo-transductors), have been used for locally transforming impinging light into an electric signal. This setup is known as a "pixel". Pixels arranged into matrices are capable of generating an image. For the present work, one needs to understand how the signal goes from photo-generation to a digital image. In addition, the reader should know the behavior (operation) of the basic pixel types.

2.1.1. Typical system architecture

Classic CMOS image-sensors typically relate to a matrix of pixels. Each pixel transforms the local impinging light into an electric signal. In this work, we refer to an imaging-system as the combination of the image-sensor (including the pixels matrix), the optics that set how light rays are directed to pixels, and the read-out that outputs a digital-image. Another aspect is how such signal goes from the pixel to next stages. Indeed, this signal corresponds to a photo-generated electric-charge stored into a capacitance -at each pixel. This charge gives rise to an output voltage for each pixel, which directly relates to the amount of impinging light -into each pixel. For recovering the whole image, the read-out stage, after the imaging-part, has to output the voltage read at all pixels. Nevertheless, later stages normally are digital-based, whereas voltage-signals from pixels are analog values. Then, after reading each pixel voltage, the image-sensor has to "transform" analog values to digital ones (e.g. the analog-to-digital conversion). This process can be achieved in several manners, as shown in Figure 1.

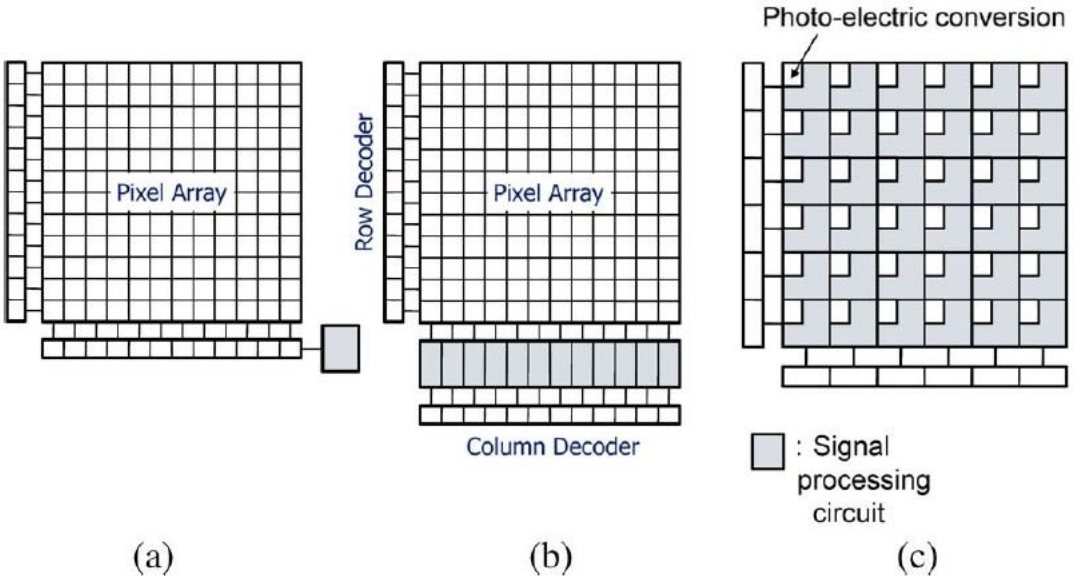


Figure 1 : different approaches for pixels pre-processing and/or read-out. "(a) serial architecture; (b) column-parallel architecture; (c) pixel-parallel architecture." (Takayanagi and Nakamura 2013)

In Figure 1, authors (Takayanagi and Nakamura 2013) explain three different image readout and/or pre-processing schemes. Figure 1 takes into account signal-processing and read-out, but for now we take only the read-out into account (letting more complex pre-processing types for section 2.2): one possibility (Figure 1 a) is to read pixels one by one, while using only one unit for carrying-out the A-D conversion. Here, transistors are used as switches for selecting which pixel value goes to the ADC. Typically, a digital controller circuit selects a row of pixels and allows them to store their values in a column analog part. This part stores, "de-noises" and amplifies pixel values. Then, another digital controller-circuit selects these pixel values, one by one, and sends them to the ADC. Another possibility (Figure 1 b) is to have one ADC per column: a digital-controller selects only the row to convert, and then all pixels of the selected row are read in parallel (with a stack of parallel ADCs). This architecture is the current one in industry of image-sensors. Another possibility (Figure 1 c), is to perform the A-D conversion at the pixel level by implementing one single ADC for each pixel. Notice that the name "Signal processing circuit", used in Figure 1, illustrates that more (pre-) processing can be performed near sensor (or matrix of sensors). That can be done even without and/or beyond the standard A-D conversion. In next section, we go a bit deeper into the inner structure of pixels, and we briefly explain several of the most common variants.

2.1.2. Pixel types

In previous part, we have cited the overall (frame-based) image-sensor architecture(s). Here, we go deeper into detail about pixel configurations and the main aspects for choosing amongst them. Figure 2 shows one simple pixel: it consists in a p-n junction in reverse-biasing conditions, a RS (row-select) switch, and a column bus (shared by all pixel on the same column). The reverse p-n junction is the photo-sensitive area (i.e. where impinging photons give rise to photo-generated electron-hole pairs). Holes drift to ground, and electrons cumulate in the junction capacitance. This charge generates an electric signal that is temporarily stored in the junction-capacitance until the read. For reading one

pixel, the row is selected by the RS switch, while the corresponding column bus is addressed, for instance, by a column-decoder. For this particular case (called the passive pixel sensor PPS), when the signal is read, charges “escape” from the junction capacitance, and thus the reading process is “destructive”. The PPS has the advantages of being simple and compact, thus favoring the easiness of higher fill factors. Nevertheless, it typically suffers from poor performances (e.g. regarding noise) respect to later topologies (Theuwissen 2007).

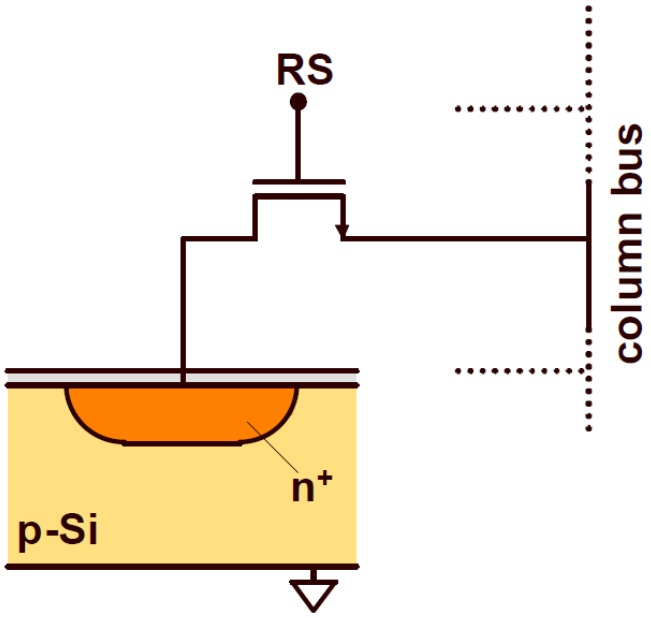


Figure 2 : illustration of the passive pixel in CMOS technology (Theuwissen 2007).

After the PPS, CMOS imaging was improved with the introduction of the so-called “Active Pixel Sensor” or APS, illustrated in Figure 3. We base our explanation from (El Gamal and Eltoukhy 2005), which gives a good overview. The term “active” comes from the amplifier (buffer) between the electric signal (the photodiode voltage) and the column bus. In this case, the readout is non-destructive. Moreover, one reset switch controls when photo-generated charges, which are stored in the junction capacitance, are discarded. The intermediate buffer-amplifier adds 1 transistor respect to the PPS, and the Reset adds another extra transistor. This is why this pixel configuration is known as the 3T-APS. This configuration allows improving the SNR respect to the PPS (El Gamal and Eltoukhy 2005), but still presents an important noise contribution from the “reset-noise”. In order to circumvent that, a forth transistor allows performing the so-called “correlated-doubled-sampling (CDS)” (El Gamal and Eltoukhy 2005). The 4T-APS is shown in Figure 3, bottom. Then, this pixel is known as the 4T-APS. This fourth transistor is, however, less conventional respect to the others in the pixel, since one of its terminals can be made of the so-called “pinned-photodiode” (El Gamal and Eltoukhy 2005). The pinned photodiode is the current industrial-standard pixel structure. Moreover, its dark-current is less important with respect to non-pinned junction. In addition, and regarding Figure 3 (bottom), signal TX allows splitting the photo-generated signal from the reading node FD. Thus, the reset signal is sampled at FD (almost) immediately before the photo-generated signal “passes” to FD (thus enabling CDS). This readout process is also non-destructive. Moreover, CDS can be set aside for the 4T-APS if a “global-shutter” readout scheme is desired. Until now, pixels are read “one by one”, but photo-current-integration is still happening during read time as well (a method known as the “rolling-shutter” reading scheme). In general, this effect is not desired. For instance, one way for letting away the rolling-shutter technique is by passing all photo-generated charges to the sensing node FD at once. That means that

photo-current-integration stops at the same time for all pixels. This strategy is the so-called global-shutter in the literature.

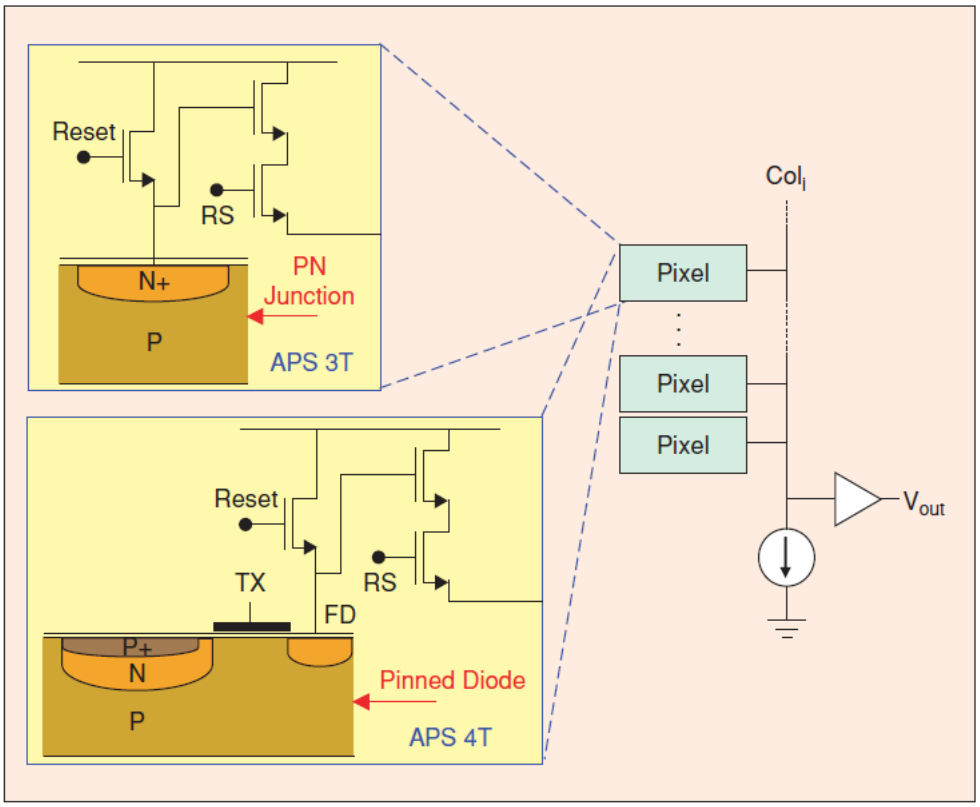


Figure 3 : illustrations of the active 3T (top) and 4T (bottom) pixels (El Gamal and Eltoukhy 2005).

The last example we present is the 1.75T-APS (Figure 4). We base our description from (El Gamal and Eltoukhy 2005). This pixel allows performing neighboring-pixels-binning and at pixel level. Binning is important for us, since we want to assess the optimal image size for object localization / classification, and the impact of image binning on algorithms performances (e.g. object localization).

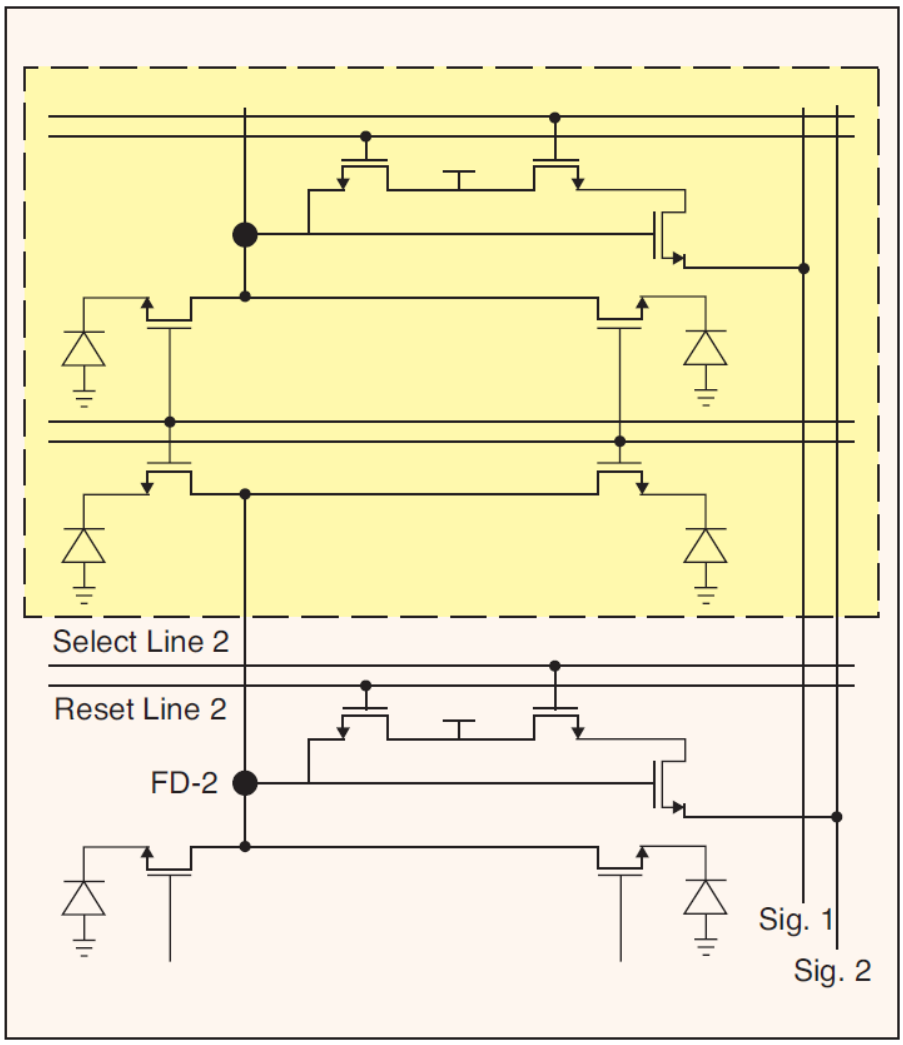


Figure 4 : illustration of the in 1.75T architecture (El Gamal and Eltoukhy 2005).

In next section, we explain some important image-sensor design parameters that we mention often during this work.

2.1.3. Typical design parameters

We base our explanation of typical design parameters from the work of (El Gamal and Eltoukhy 2005). (El Gamal and Eltoukhy 2005) explain several performance measures, and non-ideal-effects that are often mentioned in this work. They separate those non-ideal-effects into “temporal” and “fixed-pattern-noise (FPN)”. As its name suggests, temporal noise depends on time, and thus it can make the signal to change between successive readings (instead of keeping steady until the reset). On the other hand, FPN does not depend on time, but on space: it corresponds to differences in amplifiers offsets and gains across (for instance) different pixels. In addition, (El Gamal and Eltoukhy 2005) explain that temporal noise sources are (without being exhaustive) “shot noise, pixel reset circuit noise, readout circuit thermal and flicker noise, and quantization noise” (El Gamal and Eltoukhy 2005).

(El Gamal and Eltoukhy 2005) also mention important design parameters, such as the "signal-to-noise-ratio" (SNR) and the "dynamic-range" (DR). The SNR is 10 times the base-10 logarithm of the ratio of signal power and noise power:

$$SNR = 10 \cdot \log\left(\frac{P_{sig}}{P_{noise}}\right)$$

Equation 1

Where P_{sig} is the signal power and P_{noise} is the noise power. The higher the SNR, the greater is the signal respect to the noise. Moreover, the DR states the range of illumination (photons) that the image-sensor can collect and read from each pixel, settling a minimum and a maximum value. Mathematically the DR is 20 times the base-10 log of the ratio of maximum photo-current and the minimum photo-current (El Gamal and Eltoukhy 2005). The last parameter cite here from (El Gamal and Eltoukhy 2005) is the "spatial resolution".

So far we have mentioned classic architectures and parameters of image-sensors. In next section, we introduce newer ideas, which try to include more complex signal (pre-) processing schemes.

2.2. Smart CMOS image sensors

Smart-image-sensors differentiate from classic approaches due to the inclusion of integrated signal (pre-) processing. These kind of devices go beyond the typical scheme of in-pixel buffer-amplifier, column read/amplification and signal A-D conversion. The aim of integrated "smart" capabilities can be further improving design parameters (SNR, DR, etc), and at the cost of an increasing on-chip complexity. Another objective can be integrating ("embedding") artificial-intelligence algorithms, in order to optimize performance at the system level, e.g. allowing for higher speed and/or attaining lower power consumption. This work is concerned by the second case scenario, related to embedded/integrated artificial intelligence for computer vision applications.

There are several reasons for which "integrated-smart" capabilities are attractive. In the context of internet of things, distributed sensor networks, mobile robots, wearable devices, etc, devices have a limited battery that should last as long as possible without recharging. This rises the interest for "low-power" devices, which can perform artificial-intelligence-related tasks but without "draining" the battery. Reducing power has been tackled in different ways in the literature. Some trends (without being exhaustive) are reducing inter-chip data throughput by including pre-processing steps in the same chip as the image-sensor. For instance, (Verdant et al. 2020) integrated the full object-recognition pipeline into one chip with an innovative (low-power) read-out. Other strategies have been optimizing the ADC by reducing its bit-depth output, and by changing the image codification as done by (Young et al. 2019). In addition, other works have tried performing more complex computations in the analog domain (so the ADC workload is significantly reduced), such as CNN-layers, like in (J.-H. Kim et al. 2019).

In this work, we focus on two main types of smart-image-sensors: namely, frame-based (FB) image-sensors and event-based image-sensors. So far, concepts introduced apply for FB image-sensors, whereas event-based image-sensors are just briefly explained in sub-section (2.2.2). We go more into details about event-based-imaging in chapter 7.

2.2.1. Frame-based smart-image-sensors

Frame based (FB) image-sensors are characterized by a deterministic read-out scheme. In other words, the pixel readout sequence is always the same and the frame rate is constant. Moreover, each output is an image, which is acquired by two main stages: integration, and read-out. During integration time, all pixels start sensing impinging photons. Then, during read-out, pixels voltages are read by any of the schemes shown in Figure 1. For the case of FB smart-image-sensors, a (pre-) processing stage can be included to the frame-time. In next subsections, we explain some topologies in the literature following the latter idea. Notice that integration examples we found (in relation to our work) were typically related to edges or oriented gradients extraction (and/or related pipelines).

2.2.2. Edge or Oriented Gradients extraction

Edge-extraction[1] consists in computing image intensity gradients, followed by a "thresholding" operation: only pixels with a gradient magnitude higher than a certain threshold are kept. This kind of task has been included in the ADC, and some examples are (H.-J. Kim et al. 2017) and (Young et al. 2019). For instance, (H.-J. Kim et al. 2017) implemented a read-out scheme for outputting both 5-level edges data and a light-intensity-image. Furthermore, they reported a figure of merit of **70 pJ/pix/frame**. For the case of (Young et al. 2019), they proposed an ADC capable of outputting logarithmic light intensity gradients on the fly. Their proposed complete system (see Figure 5) allowed detecting objects from different classes, such as humans. The logarithmic-gradients-output from their smart-imaging-system allowed better robustness under illumination conditions (e.g. high dynamic range). That was, even under strong quantization of the log-gradients, and which they linked to the logarithm-based gradients (related to pixel-intensities ratios instead of differences). In their pipeline, logarithmic-gradients were the input for another computing stage (only simulated). There, they computed the HOG features (Dalal and Triggs 2005), which derived from the oriented-gradients image. They used those with an object-detection framework and algorithm called the Deformable-Parts-Model (P. F. Felzenszwalb et al. 2010) or DPM. The later uses a strategy based on pictorial structures, and Support-Vector-Machines, to localize and classify objects in an image.

[1] We take the convention of calling "edges" the image-gradients whose magnitude is higher to any threshold.

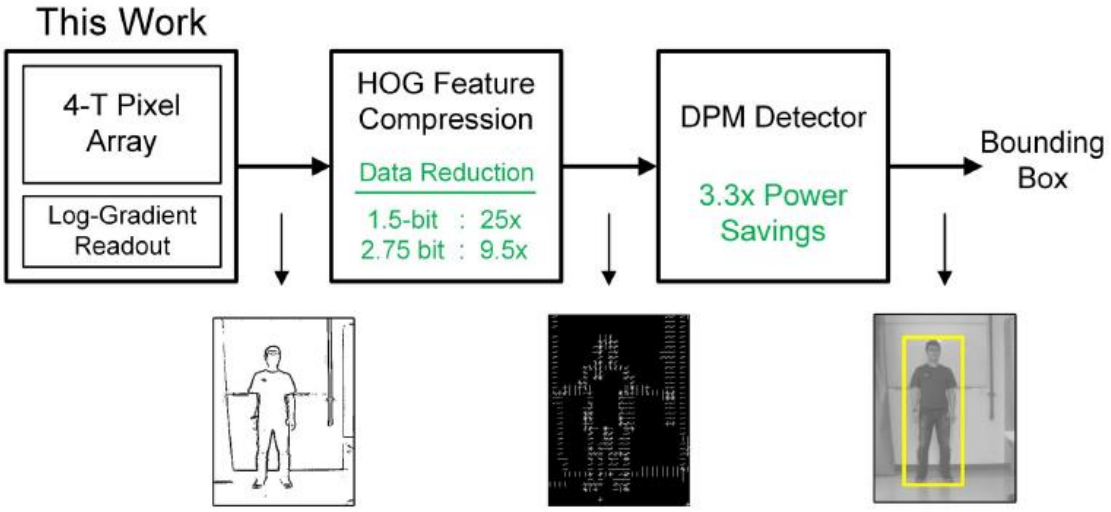


Figure 5 : logarithmic gradients object detection pipeline (Young et al. 2019).

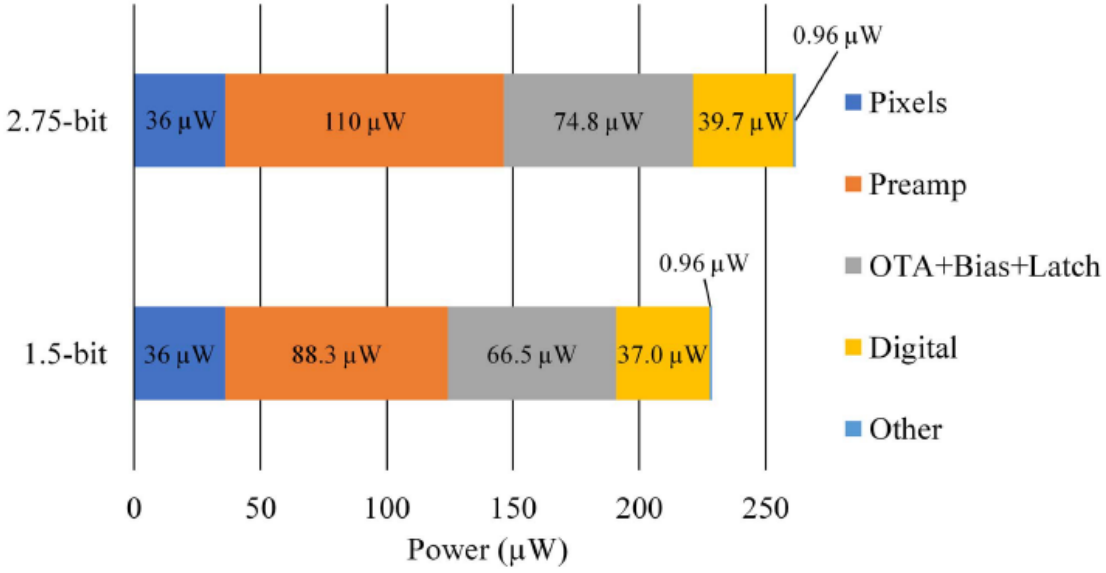


Figure 6 : distribution of power consumption for the implemented circuit from (Young et al. 2019)

Figure 6 shows their *(Young et al. 2019)* power consumption decorticated for their implementation. Notice that they implemented only the pixels-array and the logarithmic ADC (called "RDC" by them). From that figure, we observe that the orange, gray and yellow blocks are related to the log-gradients ADC, which was capable of computing log-oriented gradients on the fly. Their reported power per pixel per frame was of **99 pJ/pix/frame** (including the blue block) in Figure 6. Their log-gradients ADC is shown in Figure 7: Their circuit was capable of sampling two pixels from which the local log-gradient was obtained. The circuit used a successive approximation strategy for addressing each gradient-component magnitude and sign.

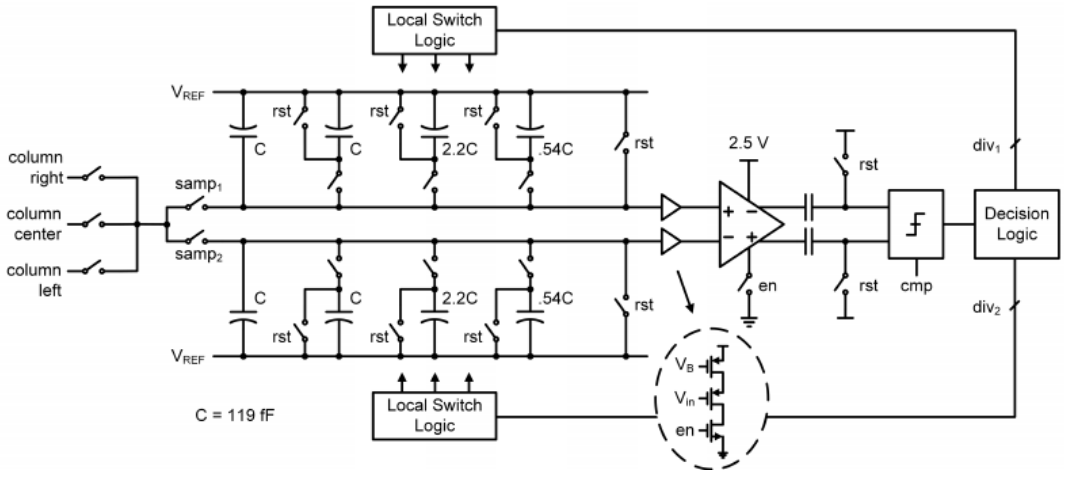


Figure 7 : circuit schematic for logarithmic gradients extraction (Young et al. 2019)

Other edge extractors have been proposed, which typically implement linear versions of the gradient approximation (instead of log), and at the bottom-of-the-column level. For instance, (Soell et al. 2016) proposed an analog edge-extraction circuitry for a matrix of 200x200 pixels that consumed (in simulation) 5.5 mW@75 fps **(2097 pJ/pix/frame**). Their work was interesting for us since they show clever circuit ideas for analog implementation (showed in Figure 8). Their architecture was based on computing both gradient components with a Sobel kernel. Then, they used both components for approximating their respective absolute value, and finally the magnitudes-sum. If the sum of both absolute-values was higher than a reference threshold, then the 1-bit output was set to "high" (indicating the present of an edge). One of the main drawbacks we found from this architecture was that no angular information was output.

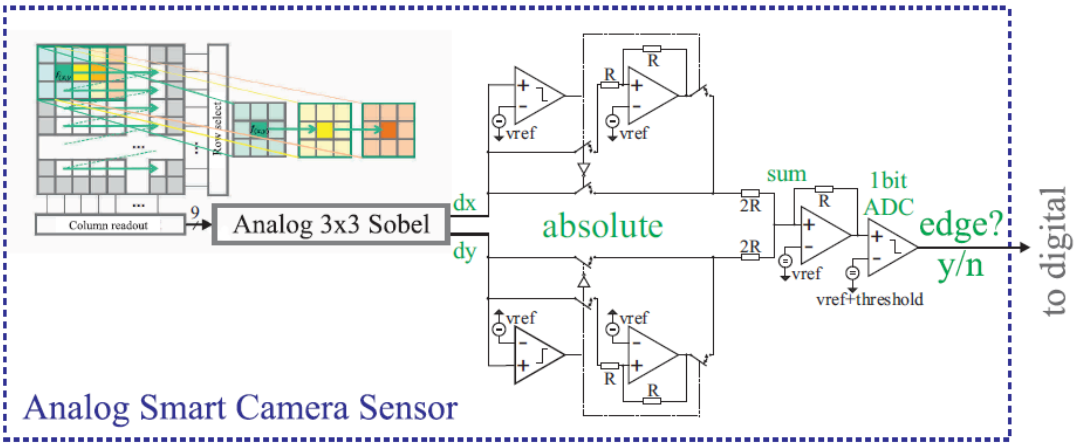


Figure 8 : circuit schematic for Sobel-like edge-magnitude-extraction (Soell et al. 2016)

Other edge extractors have computed gradients and angles in the digital side (after the ADC). For instance, (Suleiman and Sze 2014). However, as mentioned by (Omid-Zohoor et al. 2018), digital computations with the ASIC proposed by (Suleiman and Sze 2014) could imply using a standard CMOS image sensor for extracting image intensities with 8-bit resolution images, which would consume more power for the ADC than for the feature extraction. One argument can be that 8-bit intensity images

are not really necessary, and that a reduced bit depth is possible. However, (Omid-Zohoor et al. 2018) also shows that reduced-bit-depth-linear-gradients are more prone to fail under high dynamic-range conditions. Another digital implementation of the edge extraction, after the ADC, was proposed by (Jin et al. 2020): their implementation had 1920 x 1440 pixels and consumed 9,4 mW@60fps (**57 pJ/pix/frame**). However, their work does not address how to cope with high-dynamic-range conditions, neither how power would change if the angular information is needed. (Choi et al. 2014) proposed a digital block right after the A-D conversion, which can also calculate orientations. Moreover, their implementation also computes HOG features right after oriented gradients calculation. Their reported FOM was **51.94 pJ/pix/frame**, *but it cannot directly be compared* with other examples, since it also includes the HOG features generation from the oriented gradients. In the other hand, they did not take into account high-dynamic range optimizations.

Indeed, when comparing the work of (Soell et al. 2016) (2097 pJ/pix/frame) with respect to (Jin et al. 2020) (57 pJ/pix/frame) and (Young et al. 2019) (99 pJ/pix/frame), the increased power of the analog Sobel implementation can be explained by several reasons : firstly, they read pixels several times directly from the pixels matrix, implying the usage of amplifiers able to drive the column capacitances. One possible improvement, already applied by other works such as (Young et al. 2019), is buffering pixel values that are required more than once. Moreover, the Sobel filter mask is rather complicated in comparison with, for example, the mask used by (Young et al. 2019), which only requires 2 pixels per component computation. Finally, from their publication (Soell et al. 2016), we understood that the analog Sobel computing unit was shared and not parallelized at the bottom-of-the-column level. This increases significantly the bandwidth requirements for this unit, increasing the power consumption. However, we still find the schematic from Figure 8 important since it gives an insight of implementation and potential optimizations.

Other edge extractors have included corresponding electronics in pixel. Typically, the gain happens potentially in latency or power, since signals do not have to be transmitted before being processed. For example, (Valenzuela et al. 2021) proposed a smart-pixel array capable of computing pixel differences, and in order to generate a kind of feature (related to gradients) proposed by them: the Ringed Local Binary Patterns (Valenzuela et al. 2021). However, we did not find in their publication the power related for the smart-pixels-array, and for the analog-to-digital conversion. Another example is the work from (C. Lee et al. 2015): their implementation was able to detect edges, and with a power consumption of 8 mW@30fps over a 105 x 92 pixels matrix (**27605 pJ/pixel/frame**). In addition, (Yin, Chiu, and Hsieh 2016, 14) implemented an array of pixels with PWM read-out, and with in-pixel circuitry for Imaging, edge-extraction and multi-point tracking. As an example, the edge-extraction was based on pixel-intensity-comparisons (of a central pixel with its neighbors). This comparison was made with in pixel-logic, and whose output was either 1 or 0 for reflecting the edge magnitude. Their implementation consumed 34.4 µW@2160 fps for an array of size 64 x 64, giving a FOM for edge extraction of ~**3.89 pJ/pix/frame**. Interestingly, in imaging mode, their implementation consumed 154 µW@30 fps, giving a FOM of ~**1253 pJ/pix/frame**. The latter example reflects the gain of three orders of magnitude after exchanging the image codification from intensity to edges. This work is interesting from the point of view of having a very good FOM for edge extraction mode. However, several drawbacks limit our interest about their approach: firstly, the pixel complexity, containing 26 transistors per pixel, would be challenging to scale to bigger resolutions, and could show very poor fill factors. Secondly, the way edges are extracted reflects a rather simplistic logic, which is not justified to work in more complex computer vision algorithms. Finally, they do not output orientation information if an edge is detected.

One interesting question is how reported FOMs compare to a FPGA implementation. We found an example of a Canny Edge Detection algorithm implementation on FPGA (J. Lee, Tang, and Park 2018). The Canny Edge Detection algorithm runs on standard desktop computers with libraries such as OpenCV ("OpenCV, Canny Edge Detector" 2021). Continuing with last example, (J. Lee, Tang, and Park 2018) reported an optimized algorithm and implementation (on FPGA) of the Canny-Edge-Detection algorithm. It consumed 5.48 mW@50 fps for a UHD image (3840 x 2160 pixels, giving a FOM of **13.1 pJ/pix/frame**). This illustrates that features/edges computations is not necessary the most consuming part. Moreover, last examples show as well that analog computations are not necessarily less power consuming than, for instance, the more standard pipeline Image-array -> 8-bit ADC -> processing ASIC. One example supporting this argument is the work from (Choi et al. 2014), and which had a good FOM of ~52 pJ/pix/frame. Nevertheless, even though the work of (Young et al. 2019) has not the best FOM (99 pJ/pix/frame), they are the only example we found that address high-DR for feature extractors along with aggressive quantization, and that proves (with benchmarks) the interest for Object Detection.

From last examples, we observed certain trends that served us as starting point: firstly, low power optimizations seem to benefit most from near matrix-of-pixels integration of custom-ADCs. In addition, they benefit from specialized circuitry for features extraction. From examples we have found, we did not observe a conclusive reason to say that an analog or a digital implementation is more performant. The work from (Choi et al. 2014) made us wonder if computing the edge-extraction at the analog-side (and thus eliminating the need for 8-bit ADCs for A-D conversion of pixel-intensities) would lead to a significant gain in power. Moreover, from the work of (Young et al. 2019), we observed that column-parallel computations gave a much better significant FOM, and even for more complex processing (of log-gradients) with respect to the work of (Soell et al. 2016). In our work, we wonder if log-gradients could be implemented differently if we could implement another architecture which potentially gives a better FOM for oriented-edges-extraction. We did not find any particular interest in on-pixel edge-extraction, based on cited examples from the state of the art. Examples we found, like (C. Lee et al. 2015; Yin, Chiu, and Hsieh 2016, 5) were related to at least a subset of issues such as: relatively high power consumption, high pixel complexities, poor fill factors, and limited pre-processing capabilities because of the on-pixel size constraints.

We also observed relevant aspects regarding implemented algorithms for object localization or classification based on gradient-like features. Typically, different works choose a specific type of computer vision problem (e.g. face detection, identity verification, classification / recognition, etc…) and then they benchmark their implementation with a particular dataset (made by them or not). Nevertheless, most of the works we found only focus on processing images with only one object centered in them, like (Verdant et al. 2020). This might not be suitable for real case ("outdoors / out of lab") scenarios. In this work, we have chosen the Object Detection (OD) problem (localization + classification), since it is still challenging due to its complexity, and because it may have a wide range of applications that could be used in more specific contexts (autonomous driving, gaming, surveillance, etc…). The work from (Omid-Zohoor et al. 2018) (Young et al. 2019) took such approach, and they also chose OD for benchmarking the output from their imager. They used an algorithm called the Deformable Parts Model (P. Felzenszwalb, McAllester, and Ramanan 2008) (DPM), which is further explained in section 2.3.2. Nevertheless, in our work we wonder if there is a better pipeline respect to the DPM model, which, for example, would be suitable for more modern algorithms such as low-power CNNs.

Coming back to the work from (Verdant et al. 2020), their work is interesting for us, since they implemented an end-to-end image-acquisition and object recognition on chip. For the image

acquisition, they changed the standard image read-out for one that they called “Fastscan” (Verdant et al. 2020), and then they fed the image to a quantized SVM to compute binary classification “on the fly”. Their reported figure of merit for image acquisition plus object recognition was **53-59 pJ/pix/frame** for “intra-frame processing” (Verdant et al. 2020). Thus, they achieved a good figure of merit even with respect to works that only include the feature extraction. However, there are questions that, from our point of view, are yet to be clarified: firstly, the problem they solve (object recognition) is significantly simpler that problems tackled by, for example (Choi et al. 2014), and (Young et al. 2019). That is, since they targeted multi-scale and multi-class object detection. Then, it is not clear if the good FOM reported by (Verdant et al. 2020) comes from their read-out scheme and quantized SVM implementation, or just from the more simplistic machine learning problem. Secondly, and related to last sentence, there is not enough insight in the dataset they used. Therefore, we do not know how difficult the dataset is (e.g. presence of high dynamic range, variation in object’s position, occlusion, etc…). Finally, it is not clear if the edge-extraction performed on-the-fly can actually profit from the full imaging-system-DR of 36 - 88.3 dB (Verdant et al. 2020), as targeted for example by (Omid-Zohoor et al. 2018) with their logarithmic features.

So far we have cited examples for edge-extractors. Those architectures are part of a bigger family of features called in the literature “hand-crafted-features”. That is, they are human-made and tuned. The opposite corresponds to algorithms that only “impose” shapes of operations performed on data, but they do not set fixed coefficients that appear during the (pre-) processing. Instead, many of those coefficients (and the logic behind them) are “learned” under a data-driven training-process, known in the literature as “supervised learning” and explained in (Khan et al. 2018) (i.e. the algorithm is trained by using example data, where each example is labeled with ground-truth annotations). In next subsection, we cite several examples of smart-imaging-systems which implement one specific kind of trained-algorithms: the so-called “Convolutional Neural Networks”.

2.2.3. Embedded CNN-like features extraction

(J.-H. Kim et al. 2019) proposed a smart-imaging-system including a convolutional neural network (CNN). CNNs typically require multiply-accumulation (MAC) operations for subsequent cross-correlations of different kernels with the image. Moreover, the need memory access for kernel-coefficients (weights) loading and for intermediary operations. Finally, they require units capable of performing the activation-function (e.g. ReLu). In their work, *(J.-H. Kim et al. 2019)* implemented a CNN distributed between the analog and the digital domain. One particularity of such approach, is that no ADC is needed.

Figure 9 illustrates the difference between more classic face recognition approaches and theirs. The ADC is not present, and instead, an analog unit computes the first convolution stage. The output is a 2-bit low-level-features map (after a ternary quantizer), which is used by two CNNs in the digital domain. One CNN is “always-on”, and it is in charge of finding faces. In one face is detected, then another CNN is “woken-up” for the identity verification process.

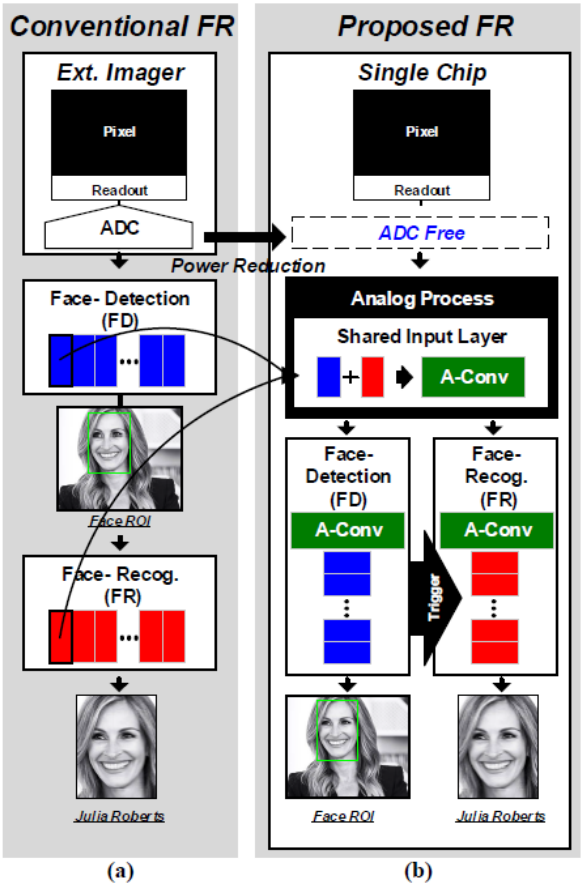


Figure 9 : "(a) Conventional face recognition system (b) Proposed recognition system" (J.-H. Kim et al. 2019).

Table 1 : comparison of the hybrid-CNN implementation and previous works (J.-H. Kim et al. 2019).

	JSSC'17 [4]	JSSC'18 [5]	This Work
Technology	TSMC 40nm	Samsung 65nm	Samsung 65nm
Algorithm	FD: Haar-like FR: PCA+SVM	FD: Haar-like FR: CNN	*FD&FR : Analog-digital hybrid CNN*
Accuracy	81% @ 32-class in LFW	97% @ whole LFW	*96.18% @ whole LFW*
Resolution	HD	QVGA	QVGA
Power	23mW	0.62mW	*0.6198mW*

Table 1 shows that the work from (J.-H. Kim et al. 2019) is viable of face detection and identity verification at 96,18 % in their tests. Moreover, their implementation only consumed 0,6198 mW@1 fps for 320 x 240 pixels, from which 10.17 – 18.75 µW where due to the analog multiply accumulation unit. *The total analog part (imaging + convolution)* was 0,0588 mW, giving a FOM of **765,5 pJ/pix/frame**. This FOM was approximately one order of magnitude higher than the case of edge-detectors like (Young et al. 2019). Moreover, we observe, from Table 1, that the CNN implementation was not significantly better in power respect to a similar solution based on Haar-like features (for only the face-detection). In addition, their reported accuracy was slightly lower for the CNN. For those two

reasons, we do not observe a particular interest for going deeper into trainable kernels for low-level features extraction, which have to handle memory access and functions such as max-poling, activation and MAC with variable coefficients.

In addition to last example, (Hsu et al. 2021) implemented a PWM pixel array and an analog convolution unit allowing 3 x 3 MAC operations (for applying the first layer of a CNN, or other kernels such as the ones needed for edge extraction). The architecture allows programmability of the kernel coefficients encoded in 4-bit weights (including the sign bit). The overall architecture is shown in Figure 10.

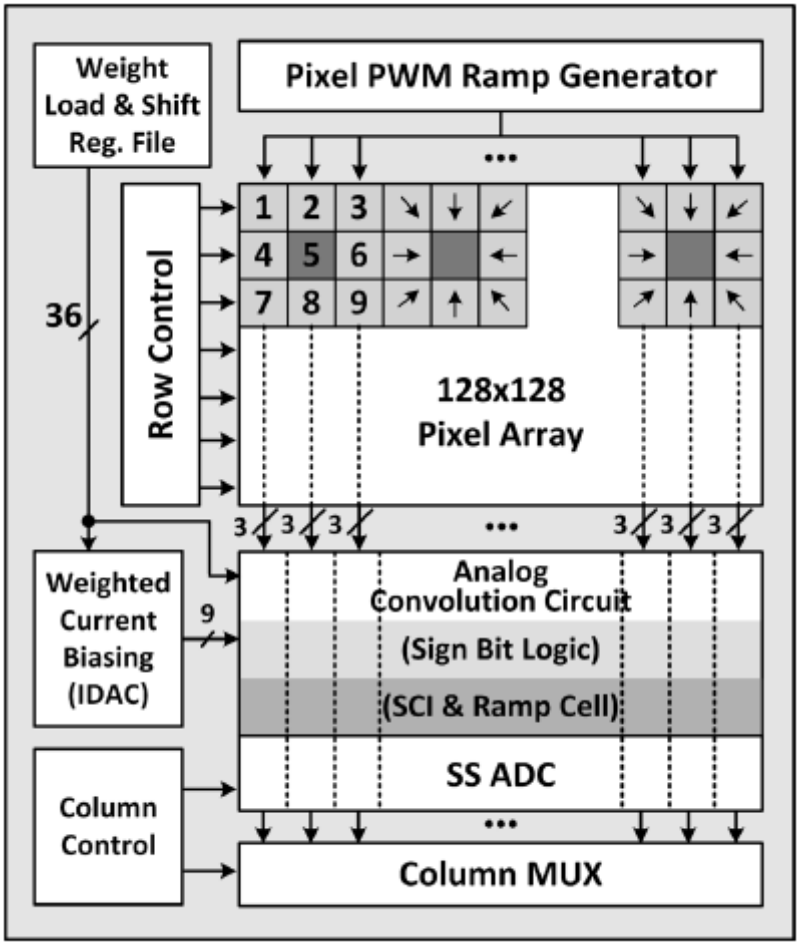


Figure 10 : system overview of the PWM pixels-matrix and programmable pre-processing (Hsu et al. 2021)

The architecture from Figure 10 allowed reading, from the pixels-matrix, 9 neighboring pixels (with a PWM scheme). Then, those pixels could be multiplied by programmable-coefficients and accumulated on the fly in the analog domain. The processing steps applied by this architecture are in Figure 11.

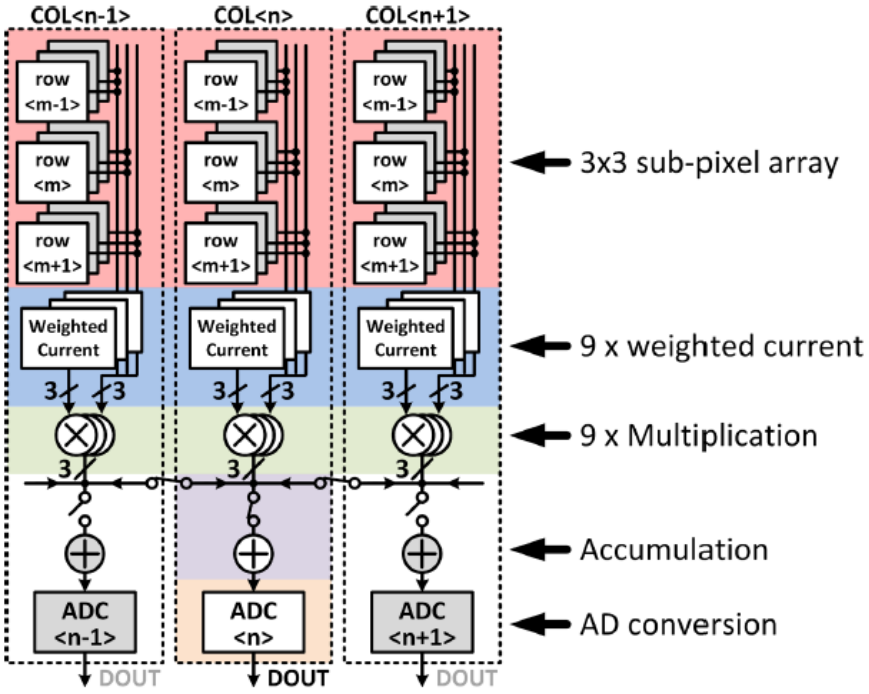


Figure 11 : diagram of the implemented pre-processing for MAC after PWD read-out (Hsu et al. 2021)

Figure 11 shows the sequence of operations performed on the image to achieve a 3x3 programmable kernel convolution with the input image. This architecture achieved a FOM of 9.8 pJ/pix/frame for imaging, and 14.8 pJ/pix/frame for the convolution, giving a total FOM of 9.8 + 14.8 = 24.6 pJ/pix/frame. This implementation achieved a significantly better (~ 10x) FOM than the work of (J.-H. Kim et al. 2019). This might suggest that a PWM read-out plus analog-convolutions with current-integration logic is a good approach for low-power feature extractors. Nevertheless, we observe the fill factor was relatively low (36 %), which probably came from their more complex pixel. In contrast, (Young et al. 2019) reported a fill factor of 60.4 % after using a more simple pixel type. Moreover, also related with the last argument, the dynamic range was 52.3 dB, which was lower than the one reported by (Young et al. 2019): 59.3 dB. Another potential drawback of this implementation is that they did not take into account big changes in scene illumination and their impact in the reported accuracy, as done by (Omid-Zohoor et al. 2018). That could suggest that this kind of linear pre-processing along with an aggressive quantization could suffer from scenes that are not uniformly illuminated.

Another example of works for integrating CNNs is given by (Bose et al. 2019): they proposed an implementation of “Pixel processor arrays” (Bose et al. 2019) for computing ternary convolutions on the focal plane. Each pixel contained basic hardware, as illustrated by them in Figure 12, for computing addition, subtraction and bit-shifting. In their work, they explain how to represent ternary CNN convolutions with the limited hardware available.

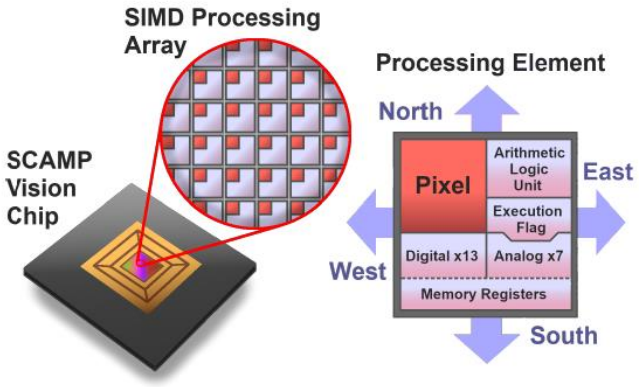


Figure 12 : illustration of the in pixel-MAC architecture (Bose et al. 2019)

In last paragraphs we have cited different works that show interesting applications of integrated image pre-processing. One common trend is the increased circuit complexity, either at pixel level or at the periphery. In next section, we cite examples that use 3D-IC technologies for integrating relatively complicated and highly parallelized pre-processing algorithms.

In our work, we preferred to tackle hand-crafted near-matrix-of-pixels feature extractors before going into more complicated implementations without having clear if they are really justified.

2.2.4. 3D-IC Smart Image Sensors

(Millet et al. 2018) implemented a 3D-IC smart-image-sensor chip with two tiers (layers). Their microfabrication technology allowed them to parallelize communications between the two layers, as shown in Figure 13.

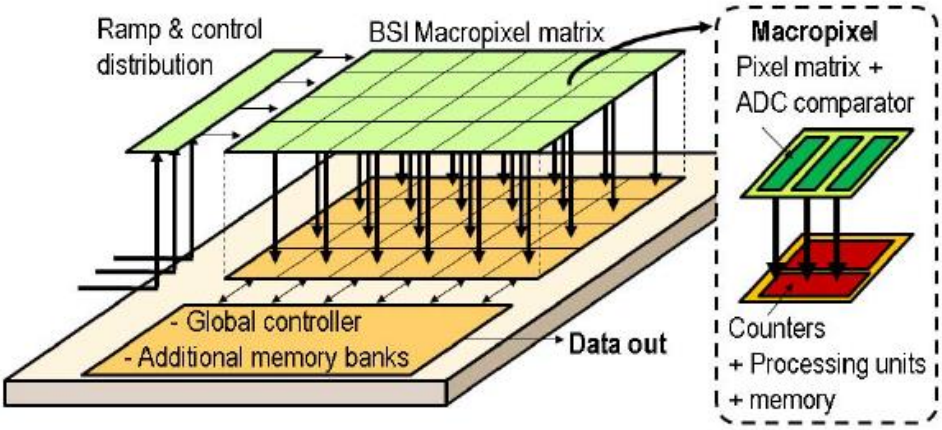


Figure 13 : illustration of the 3D-IC smart-image sensor (Millet et al. 2018)

Figure 13 shows the system overview implemented by (Millet et al. 2018): the top layer (tier) included an array of what they called “macropixels”, which include a 16 x 16 arrays of pixels, plus an

ADC. Each macropixel could communicate with a processing unit located right underneath, so local operations could be performed with improved latency and with a high degree of parallelization. Figure 14 shows each processing-unit-architecture. Indeed, this same Figure 14 indicates that one unit was dedicated for interfacing the pure computational unit (right) with the pixels on the top tier. Their processing unit (called "PE" for processing element), contained hardware capable of 8-bit computation of rather complicated functions (for near-sensor ICs) such as sinus, cosine and square root.

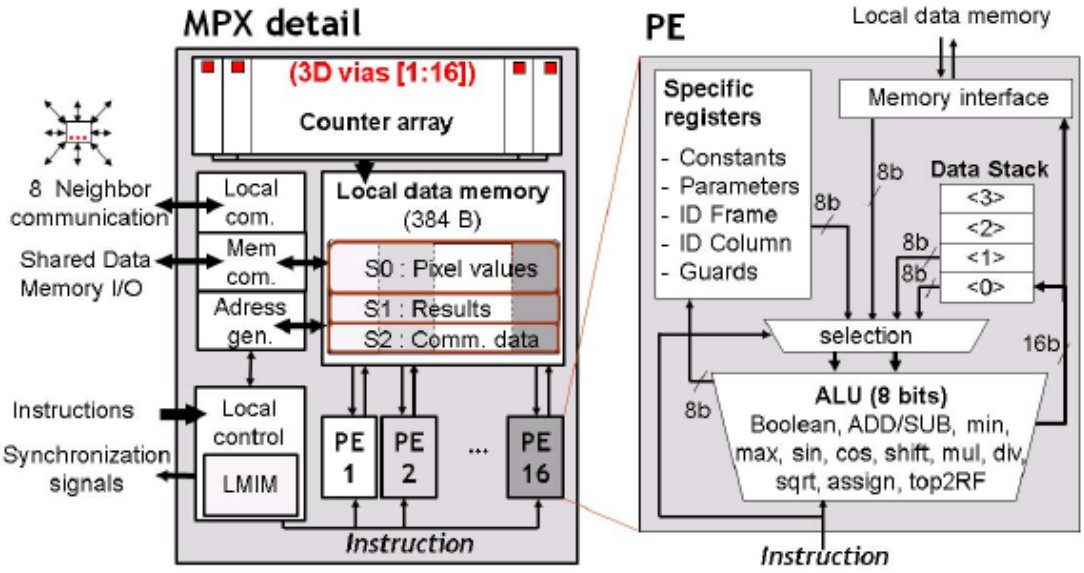


Figure 14 : diagram of the units implemented in the bottom layer (Millet et al. 2018)

The implementation from (Millet et al. 2018) allowed interesting figures of merit, especially regarding latency : for an array resolution of 0.05 Mpixels with a 9 bit-depth encoding and a frame rate of 5500 fps. However, power-wise, the consumption for this arrangement was 720 mW, giving a FOM of ~**166.5 pJ/pix/frame**. This figure of merit is significantly less attractive than other works already cited before, also considering that it only takes into account image acquisition. Moreover, for low power applications, it is not clear for us how this system would benefit of such elevated frame rate without prohibitive power consumption from data I/O.

(Suarez et al. 2012) proposed and simulated a 3D-IC staked architecture for image acquisition and feature extraction. They present optimized mixed-signal (top-tier) and digital (bottom tier) electronics, for targeting complex algorithms such as interest-points detection. Interest-points is a concept in computer vision, related to zones (points) in the image (along the scale-space pyramid) that can be matched even if they re-appear later in a video sequence at another position. Those interest points are found with known algorithms based on handcrafted descriptors. In their work, (Suarez et al. 2012) mentioned the Harris, Hessian and Difference of Gaussians descriptors. In their architecture, the top tier was in charge of image acquisition and generating (in parallel, in mixed-signal) a Gaussian pyramid. The bottom tier was in charge of computing the corresponding descriptor and of detecting the interest-points based on the corresponding local descriptor value. They reported a simulated value of **3600 pJ/pix/frame** for image acquisition, Gaussian pyramid generation and A-D conversion. Without A-D conversion, their reported FOM was **50 pJ/pix/frame**. The relative difference between these two FOMs suggests that removing or optimizing the A-D stage is critic for attaining figures of merit comparable with 1 layer smart-imaging architectures for low power. Again, we observe that the gain

from a 3D-IC implementation was the system being able to reach low latencies thanks to the high parallelization, of 50 µs for Gaussian pyramid generation from a QVGA image.

(Eki et al. 2021) implemented a two layer stacked IC with the imaging acquisition section on the top tier. In addition, they implemented a dedicated CNN computation unit on the bottom. They showed that their system was capable of computing a quantized version of Mobilenet (with latencies of 3.1 ~ 3.4 ms). That shows the applicability of a relatively complex CNN for low-power and embedded computer vision. However, as in previous examples, the power consumption due to the imaging part was considerably higher, and with respect to the computing part. For instance, we can take into account the energy related to 2 convolutions, and with kernels of size 3x3 (Sobel) like in (Soell et al. 2016). For a chip of 4056 x 3040 pixels, their reported power is 278.8 mW@30 fps, giving a FOM for the image acquisition of 753.7 pJ/pix/frame. Their reported TOPS/W ("tera-operation per Watt") was 4.97 TOPS/W. We consider that the convolution of one single kernel of size n x n = 3 x 3, and with an image of size h x w = 4056 x 3040, gives a total amount of multiply-accumulation operations (MACS) of $MACS = n^2 \cdot (h-2)(w-2) \approx 110.8$ mega-operations. We can multiply this number by two (considering that edge detection typically implies 2 kernels, one for each gradient component) and divide it by the TOPS/W. That gives the power for convoluting 2 kernels with one single frame, corresponding to ~22.3 µW/frame. For a frame rate of 30 fps, the corresponding FOM at the same image size is ~54.3 pJ/pix/frame. The total estimated FOM for image acquisition and convolution with 2 kernels of size 3 x 3 is **~808,0 pJ/pix/frame**, from which ~93 % comes from the image acquisition. Indeed, this FOM is similar to the one from (J.-H. Kim et al. 2019). However, in their case, the analog part consumed ~10x less than the digital CNN processor. In addition, (Hsu et al. 2021) also obtained a better FOM thanks to the PWM image acquisition scheme. Those two examples suggest that the work from (Eki et al. 2021) could be further improved by changing the column ADCs for an analog first stage convolution (which could potentially reduce the FOM due to acquisition only).

From 3D-IC staking technology examples we have cited so far, we observed that there is no clear reason, power-wise, for 3D-IC stacking. As we cited before, figures of merit are worse than single layered examples in the SoA. It is worth noticing that some works from 3D-IC staking have reported high-frame rates, probably thanks to the strong parallelization. Nevertheless, it is not clear for us how such high frame rates would be exploited in the context of low-power computer-vision-applications. Another potential interest, is the possibility of embedding more complex CNN architectures such as the case of benchmarks presented by (Eki et al. 2021), and with Mobilenet + SSD. Then, this is an illustrative example of 3D-staking technologies combined with embedding more complex algorithms. Another example supporting that argument is the work from (Millet et al. 2018): they can attain more important memory storage, and more complex operations (such as cosines) in parallel. Thus, from our perspective, the point of having a 3D-IC stack is to perform on chip (pre-) processing like localization/classification algorithms. Then, only high-level (semantic) data is sent by I/O interface, and so giving a potential power gain by reducing the data-rate with the "outside circuitry".

Until this subsection we have cited examples of frame-based approaches. The state of the art presents many examples of this scheme being successfully used for computer vision purposes. Nevertheless, there is a general problem that they all share: the entire image is always acquired and processed, even if "nothing" has changed. That sets the need for a probably smarter approach, and in which only portions of the image where changes are present are processed. For instance, local processing only happens when local changes occur, instead of doing it regardless if nothing has locally changed. Ideas such as that one have inspired a family of non-classic imaging acquisition techniques: the "event-based" image sensors. In next subsection, we will explain them further and provide some examples on the SoA.

2.2.5. Event-based imaging-systems

Event based image-sensors typically codify information as a stream of asynchronous events. They do not output a stream of images, read "frame-by-frame" (by means of a rolling or global shutter), like it is the case of FB imagers. Figure 15 shows an illustrative pixel schematic (left) and functioning principle (right) of one example from those pixels: the dynamic vision sensor (DVS) (Posch et al. 2014). The pixel corresponds to three main stages: firstly, there is a reversed biased photodiode for photo-current generation plus amplification. Secondly, next stage permanently computes the approximated relative difference between the instantaneous I_{ph}, and the one sampled (as Voltage) at C_1 at the last reset. Finally, the third stage permanently computes the comparison of the absolute value of this difference (with a double threshold) with a reference (set by comparators bias). When any of the two comparators in Figure 15 switches to the active stage, an asynchronous-request-signal is sent to activate the event-read. Figure 15 (right) shows how those events (codified as spikes) are sent each time the voltage at the sensing node V_{log} changes (up or down) sufficiently. Moreover, those spikes can be accumulated during a time frame to render an image (such as the one represented in the same Figure 15).

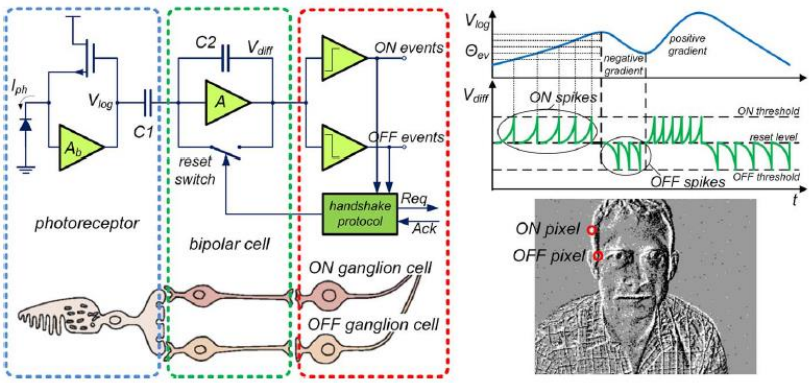


Figure 15 : illustration of the principle of functioning of the dynamic vision sensor (Posch et al. 2014)

Continuing with last idea, we no focus this paragraph on the work from (Lichtsteiner, Posch, and Delbruck 2008), related to Figure 16. This figure shows a typical architecture of Dynamic Vision Sensors: an array of pixels are constantly sensing the relative light intensity variation. When any of them send an asynchronous request signal (a spike), blocks handshake logic and the arbiter are in charge of "reading the event". Moreover, they also manage events coming approximately at the same time, and send back acknowledge signals (for each pixel separately) that trigger the local reset.

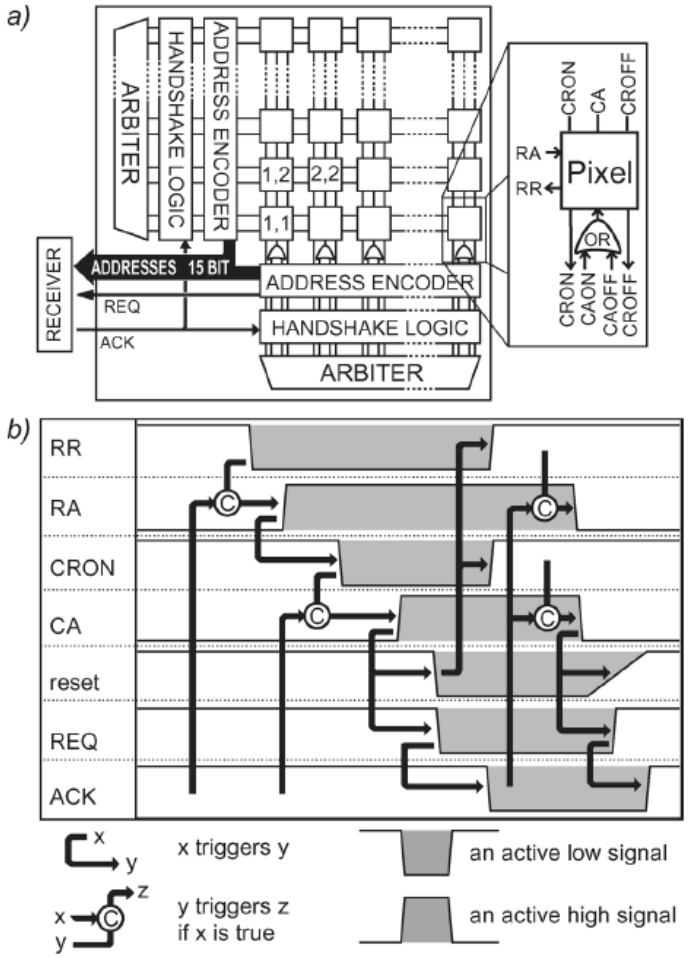


Figure 16 : example of the event-based read-out "(a) Block diagram. (b) Timing for a communication cycle for a single ON event…" (Lichtsteiner, Posch, and Delbruck 2008)

Event Based Vision sensors aiming to recover temporal data (such as the DVS pixel and architecture from Figure 15 and Figure 16) are not the only non-standard scheme that has gained popularity in the SoA. Other imaging acquisition techniques can benefit from the event based codification for light intensity measurements. One example is the "time-to-first-spike" (Guo, Qi, and Harris 2007) imager, or TTFS. This kind of imager codifies each pixel-light-intensity as the time for the pixel-signal to reach a certain threshold. The principle of functioning is illustrated in Figure 17. The pixel-signal is the voltage at the sensing node resulting from the photo-current integration after a global-reset. Notice that, as the case of FB imagers, there is a global reset. However, pixels do not output a voltage signal which then is transformed to the digital domain by means of analog-to-digital conversion. Instead, they just send a request signal (a spike) to an arbiter circuit. Then, dedicated circuitry takes into account the individual address and spiking time for converting such time to lighting intensity.

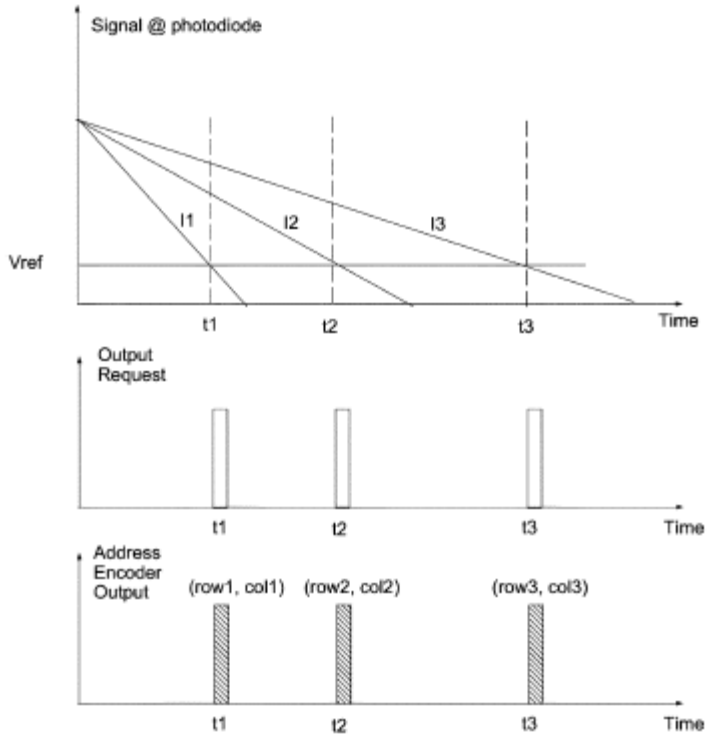


Figure 17 : principle of functioning of the time-to-first-spike pixel (Guo, Qi, and Harris 2007)

Examples mentioned before for event-based image-sensors made us wonder the following: is it still possible to embed near sensor "logic" for performing computations? For instance, is it possible to handle summations, subtractions, multiplications and divisions? Indeed, several works have proposed logic units that work with spikes. For instance, (Ravinuthula and Harris 2004) proposed electronics for performing smoothing operations on pixels (see Figure 18), or for computing pixels differences which are then compared with a fixed threshold (see Figure 19). Moreover, they explain how those can be implemented in order to perform edge detection with the output of TTFS imager.

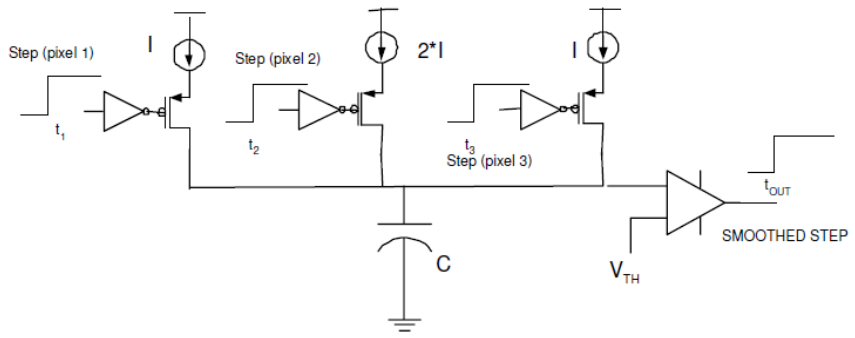


Figure 18 : example of a weighted average circuit in spike-domain (Ravinuthula and Harris 2004)

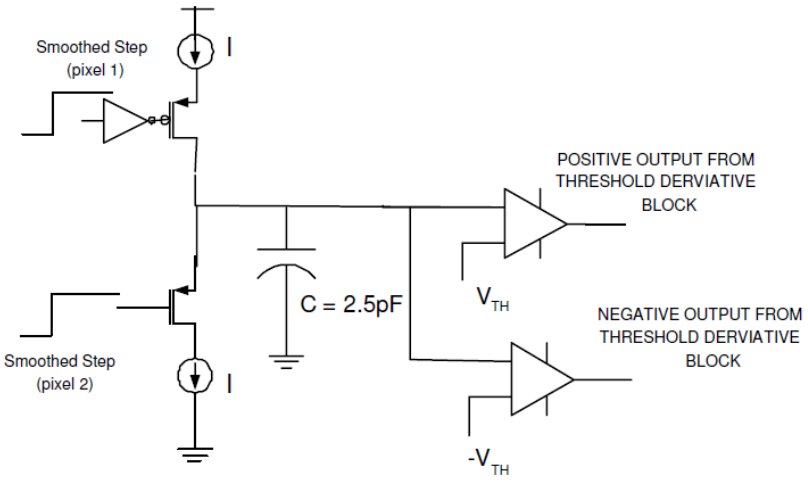


Figure 19 : example of a subtraction and thresholding circuit in spike domain (Ravinuthula and Harris 2004).

Until now, we have discussed integrated and embedded imaging-system architectures, and for low-power computer-vision/artificial-intelligence. Some of them presented IC implementations as well. Notice that this work points as well towards one specific problem of machine learning and computer vision: object detection (OD). Next section will discuss further about it, before we start making the link with sections 1.2 and 1.3.

2.3. Object detection pipelines

2.3.1. HOG and linear SVM

(Khan et al. 2018) explain that the object detection problem consists in solving to sub-problems in computer vision: firstly, there is the classification problem (labeling each object where each label corresponds to a particular object class). Secondly, there is the localization problem, which consists in giving coordinates (e.g. bounding boxes) of the different objects. In next paragraphs, we give a very general, non-exhaustive overview of the main object-detection pipelines in the state of the art.

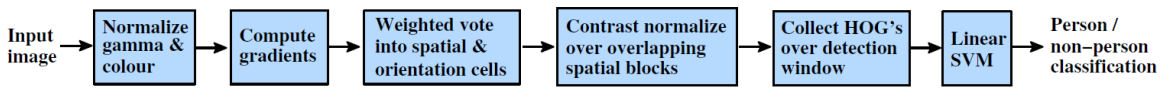


Figure 20 : example of an object detection pipeline with HOG and SVM (Dalal and Triggs 2005)

Figure 20 shows a relatively old pipeline (published in 2005), but that is still relevant for embedded cases, like (Suleiman and Sze 2014) and (Choi et al. 2014). The algorithm takes an intensity gray image from which it computes the local intensity gradients (magnitude and direction). This derived gradient map is divided into blocks called “cells” by (Dalal and Triggs 2005), and for each cell they proposed to obtain a feature called “Histogram of oriented gradients” (Dalal and Triggs 2005) (HOG). Then, a “sliding window” with a template of the object of interest (in HOG representation) is

compared in as much as possible regions of the image (scale and space wise). The template is “learned” in their case, and by the implementation of a linear support vector machine.

2.3.2. HOG and DPM

Later, (P. F. Felzenszwalb et al. 2010) presented a more complex model for object detection, which takes into consideration that objects can be “understood” or “modeled” as a collection of sub-parts. In their model, those sub-parts can move respect to the others. Figure 21 illustrates how objects are “decomposed” into “parts” for the purpose of object detection. They (P. F. Felzenszwalb et al. 2010) proposed a model which they called the “Deformable Parts Model” (P. F. Felzenszwalb et al. 2010). This model has a similarity with the purely HOG approach from (Dalal and Triggs 2005), in the fact that is uses HOG features, and it performs the detection over a “HOG space”. Nevertheless, it changes the idea of a simple template representing each object, for a collection of templates that represent each part. Moreover, sub-parts can move respect to each other. However, templates allowing sub-part matching are limited in the range from which they can move respect to the “center of the model” (called by them the “root”), and this limitation is represented by a penalty function. Thus, the detection is determined by a global function taking into account the different templates-matching, and their relative position to the “root”. This work was used indeed used for benchmark purposes by the circuit implementation from (Young et al. 2019).

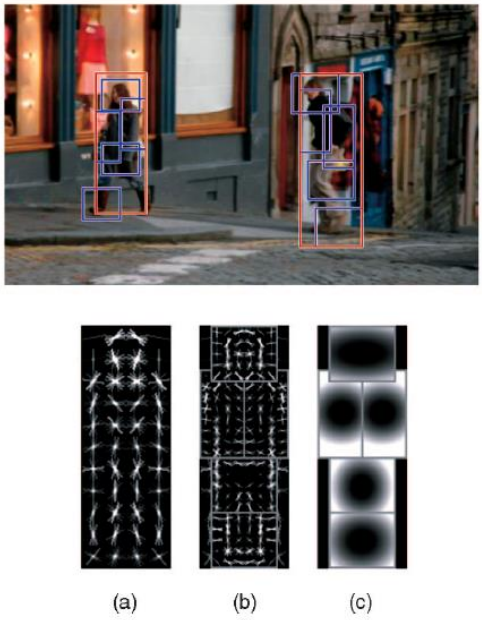


Figure 21 : illustration of the principle of functioning of the DMP model. “(a) a coarse filter, (b) several higher resolution part filters, and (c) a spatial model for the location of each part relative to the root…” (P. F. Felzenszwalb et al. 2010)

2.3.3. Fast R-CNN

In recent years, convolutional networks (CNNs) have gain popularity in computer vision, including object detection. For instance, Figure 22 presents a brief of the “Fast R-CNN” (Girshick 2015)

pipeline. It takes a colored input image, and then it takes it as input for a “Region proposal algorithm” (not shown in the figure). The region proposal algorithm outputs a series of region proposals or “bounding boxes” which are used later. Moreover, the colored image is also passed by a convolutional neural network from which a “convolutional feature-map” is derived. Then, the region proposals are projected into this feature map with an operation that they called the “RoI pooling layer” (Girshick 2015). From each RoI, there is one output from the RoI pooling layer, which is then used for classifying the region proposal and for estimating the coordinates of the bounding box. This pipeline is interesting for us since the idea of “Region proposals” can be potentially more efficient (power and computational wise) than the sliding window approach when implemented, for instance, on a 3D-smart-image-sensor.

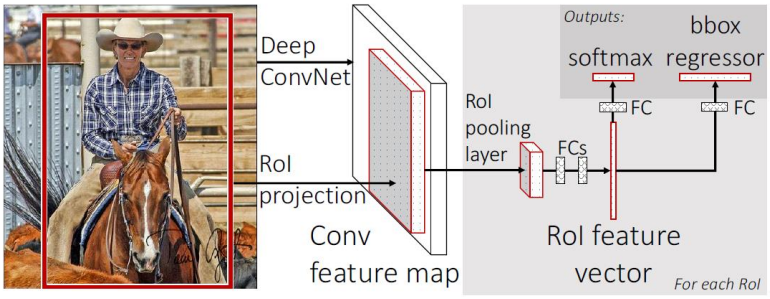


Figure 22 : diagram of the Fast R-CNN pipeline: “… An input image and multiple regions of interest (RoIs) are input into a fully convolutional networks. Each RoI is pooled into a fixed-size feature map…” (Girshick 2015)

2.3.4. Faster R-CNN

Later, (Ren et al. 2017) proposed to replace the “external” RoI proposal algorithm by a CNN called by them the “Region Proposal Network” (RPN) (Ren et al. 2017). Moreover, this RPN shared convolutional layers with the classifier stage. Indeed, the input image was passed to a series of initial convolutional layers, from which a feature map is derived. Then, this feature map is passed to the RPN, and the output from the RPN is then used for “RoI pooling” the Region proposals before passing then to the final stage: the classifier. This approach is similar to the one implemented by (J.-H. Kim et al. 2019), related to Figure 9. However, there are important difference to consider: firstly, the shared convolutional layers were replaced by a single CNN layer in the analog domain. Secondly, the RPN was changed by a face detection CNN, which could be understood as a “face-region-proposals stage”. Thirdly, their implemented face recognition stage (which would be similar to the classifier in Figure 23) was triggered only if the face was detected at a certain region of the image, and the classifier would evaluate only this specific region as input.

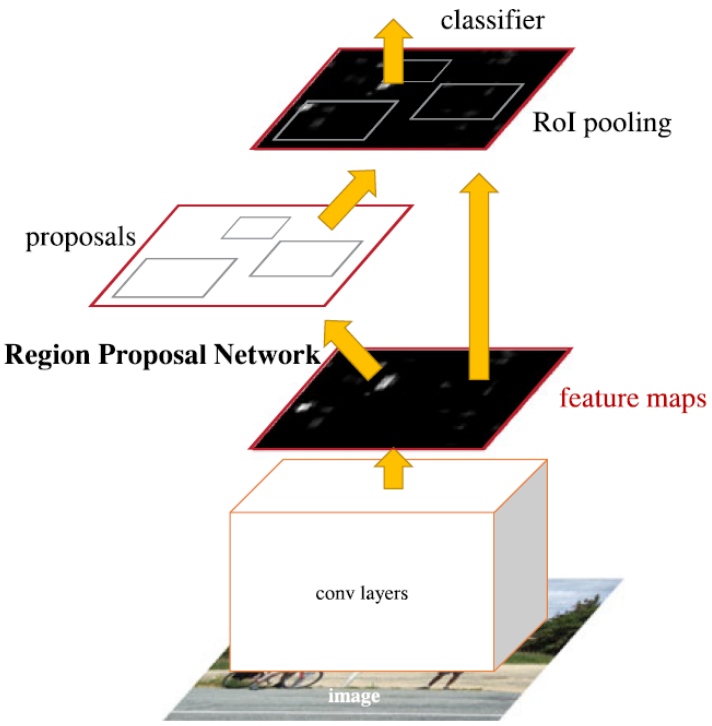


Figure 23 : illustration of the "Faster R-CNN" pipeline (Ren et al. 2017)

2.3.5. SDD and YOLO

Finally, other works, such as (Redmon et al. 2016) and (Liu et al. 2016) are other kind of strategies that do not use the region proposals method. For instance, Figure 24 shows the working principle of the work from (Redmon et al. 2016): the image is divided into equal regions, and for each region there are C different "bounding boxes". Then, for each bounding box in each region there is always a prediction of the class (or no class at all). This kind of strategy is not our first choice since it computes the classification stage in many regions (the number of image sub-regions multiplied by the number of bounding-boxes per region) in spite of if there is potentially an object or not. Then, it is less clear how to implement a wake-up or hierarchical system into which the portion expected to consume most of the power (the classifier) is used only when there is a minimal chance of finding any object.

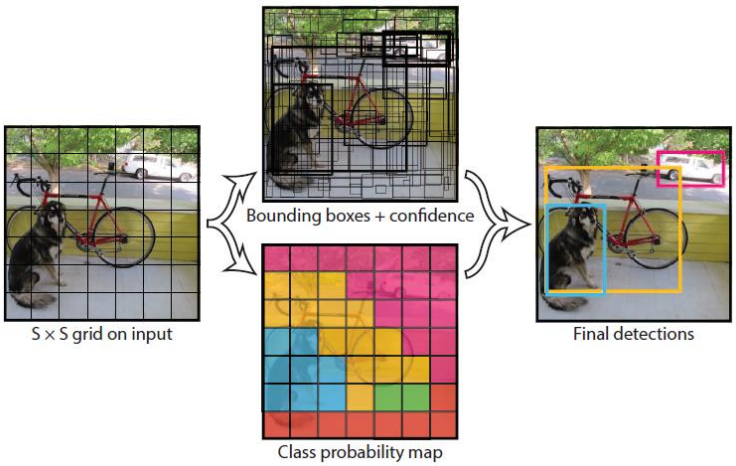


Figure 24 : illustration of the principle of the YOLO-pipeline: "…It divides the image into an S x S grid and for each grid cell predicts B bounding boxes, confidence for those boxes, and C class probabilities…" (Redmon et al. 2016).

In this section, we have cited several illustrative works of the trending algorithms for object detection, and for general purpose machines. In addition, we have cited works in the embedded vision context that get inspiration from the aforementioned pipelines. From there, we have selected to target Region Proposal Based pipelines since they could be compatible with hierarchical/wake-up mechanisms for low power, and they could reduce the classifier power at the cost/overhead introduced by the ROIs generation. For instance, our focus is to go deeper in a pipeline resembling Fast-RCNN, where we use a secondary algorithm (non CNNs-based) for ROIs-generation. We preferred to start from an algorithm that does not depend on CNNs for ROIs-generation, since it is not clear for us if the low level implementation of a CNN for ROIs would be advantageous or not. Then, one approach is to assess the best way of ROIs-proposals by studying both approaches (CNN-based with a Faster-RCNN-like pipeline, or by with non-CNN-like ROIs-generation as in Fast-CNN). Nevertheless, in this work we study only the non-CNN ROIs-generation, and we let the second case to further works. Finally, we discard the DPM model, since it seems, from our perspective, a prior (not necessarily less "heavy" in terms of computations) to CNN algorithms.

2.4. Conclusions of chapter 2

In this chapter, we have presented an overview of the theoretical background (State of the art) related to this work. We have cited illustrative works in the SoA that give us a point of departure. From there, we have made the preliminary decisions that drive the rest of this work:

1. We focus our attention in bottom-of-the-column parallel pre-processing, letting the matrix of pixels "untouched". Indeed, it allows to split the optimization of the imaging part and the pre-processing stage. Moreover, we do it in order to have simpler pixel architectures and classic read-out schemes, such as the case of the 4T-APS/1.75-APS.
2. We focus on the generation of hand-crafted features by means of a dedicated, non-programmable architecture, specialized for the kind of feature we want. Moreover, we follow the trend we observed in the SoA regarding the gradient-based features (such as variants of edge-extractors). However, we diverge from the SoA since we want to optimize the extraction of those features for the purpose of Region Proposals Generation on another chip, or in the bottom tier if 3D-IC staking is used.

3. We take inspiration from the pipeline of Fast-RCNN for achieving a good trade-off of hardware/computational-complexity and performance. We diverge from works in the SoA which tackle the OD problem by using the sliding window approach like (Young et al. 2019; Omid-Zohoor et al. 2018), or by using region-proposals (face-detection) with a CNN (J.-H. Kim et al. 2019). We use the same strategy as (Girshick 2015), which corresponds to using an algorithm for the Region Proposal generation.

In next chapter, we will present out methodology for behavioral simulations, which we use along this thesis for obtaining our results and for driving our conclusions.

Chapter 3. Our simulation Framework

In this chapter, we describe the framework developed during this thesis. Its objective is to allow smart imager system behavioral simulations in order to optimize the image-processing-pipeline design. One key aspect is that, for several specific stages (further developed in this chapter), it takes into account hardware constraints (e.g. memory access constraints, quantization, noise, suitable operations, etc). Moreover, we derive the impact of those constraints on relevant metrics related to artificial intelligence and integrated systems.

The reason why we made our own simulation framework were the following: firstly, we wanted a tool that takes into account both electronic schematics and rather complex signal processing pipelines. Such tool would be in between a specific CAD tool like Cadence and a library such as OpenCV. The second reason was that we wanted a software allowing separation of behavioral models into two main groups: the ones that are integrated, and those that would run on a standard CPU or GPU. The integrated stages would include hardware constraints, while the rest of the stages will run without considering them, but their input is the output from integrated-behavioral-models (which do consider hardware constraints). The third and final reason was that we wanted a framework capable of simulating different kinds of standard and non-standard types of imagers.

We call our framework EdgeTon ("Edge" meaning that is for Edge-AI design, and "Ton" referring to be Python code). In the next sections, we start developing the framework architecture and functionalities. After, we discuss several imager system models that will be important in the next chapters.

3.1. EdgeTon main architecture

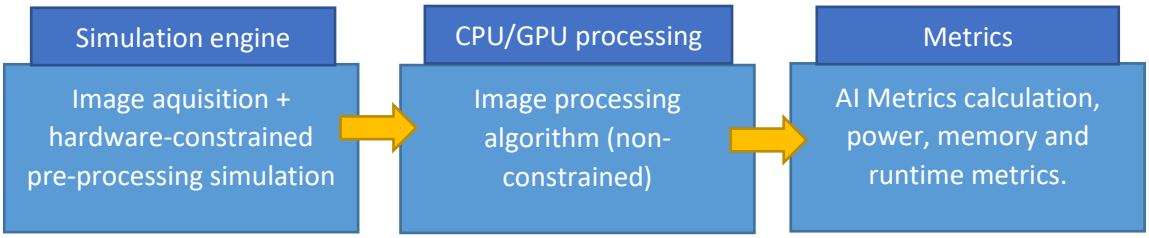


Figure 25 : main Edgeton algorithm.

Figure 25 shows the flow for any general EdgeTon simulation. Firstly, the framework allows loading and customize an imager model, which is used by the simulation engine to simulate hardware-constraints on the image acquisition/pre-processing. The output from there is later used by a standard algorithm (with a standard framework such as OpenCV or Tensorflow) to assess the algorithm performance when the input comes from a hardware-constrained/integrated image-sensor and pre-processing. Finally, the simulation results are loaded by third module, which is in charge of calculating all the metrics used for human interpretation. The figure below shows a more detailed structured of the framework, which depicts different functional blocks important for the code-implementation.

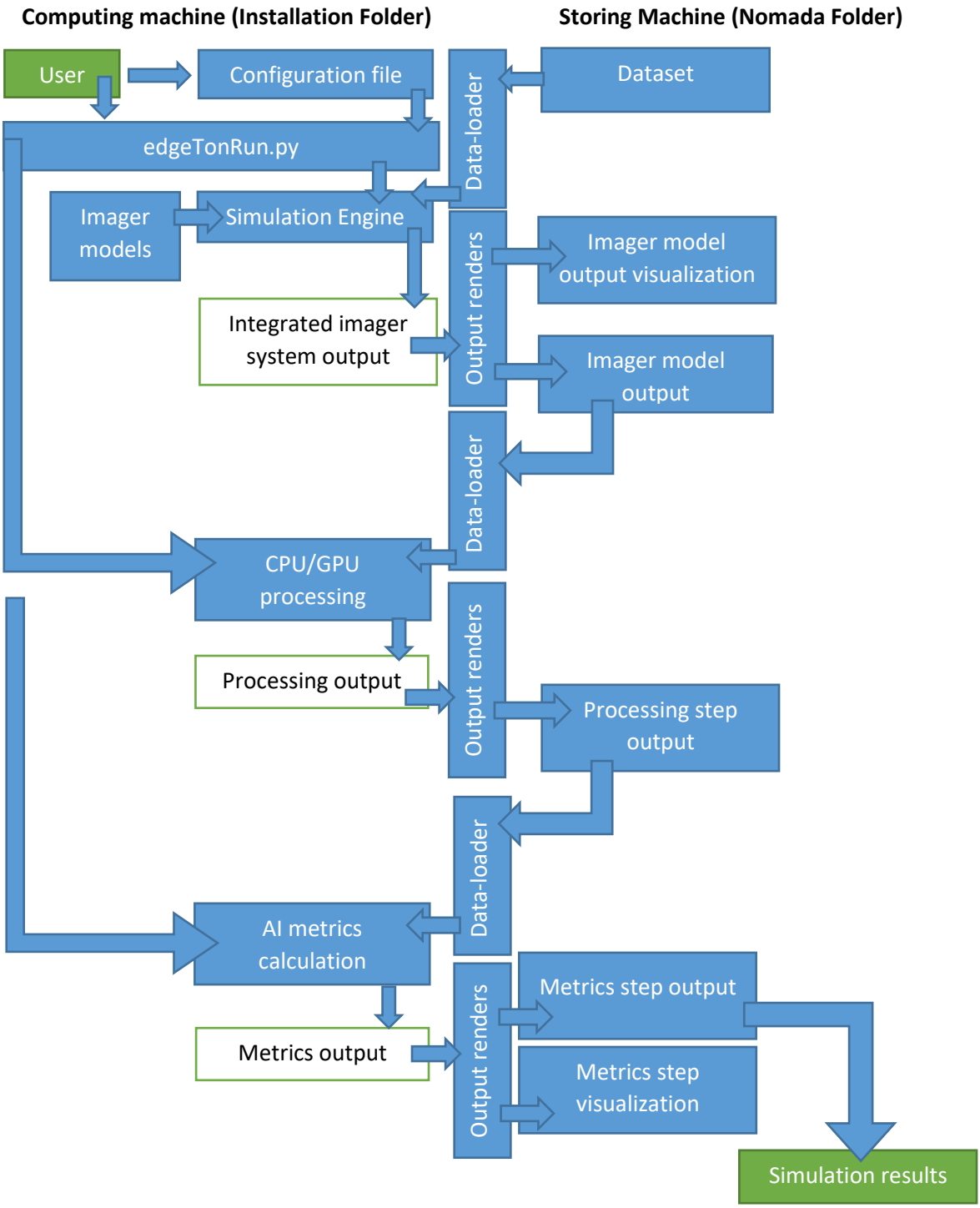


Figure 26 : diagram of EdgeTon framework architecture for simulating complete processing pipelines.

Figure 26 presents a diagram of our framework for simulating (behaviorally) smart imaging systems (integrated plus a portion potentially in another chip or device). The left side represents the code architecture, the green color representing the starting point: the user, who provides a configuration file. Then, the main program, called edgeTonRun.py, launches sequentially each portion of the complete processing pipeline.

The first stage is the simulation engine, which is in charge of simulating the part of the system that is hardware-constrained. Typically, this portion relates to the image acquisition and processing integrated near (or in) the pixel array, and only performs several initial steps in the processing. More

precisely, we make the difference (in our models, and corresponding code) between the pre-processing that happens inside each pixel, and the one that happens after pixel readout (right before or after the ADC). We refer to the first case as “deep pre-processing” (or near pixel processing), and to the second one as “shallow pre-processing” (or near ADC processing).

In the next section, we focus on the general imager model.

3.1.1. Imager models structure

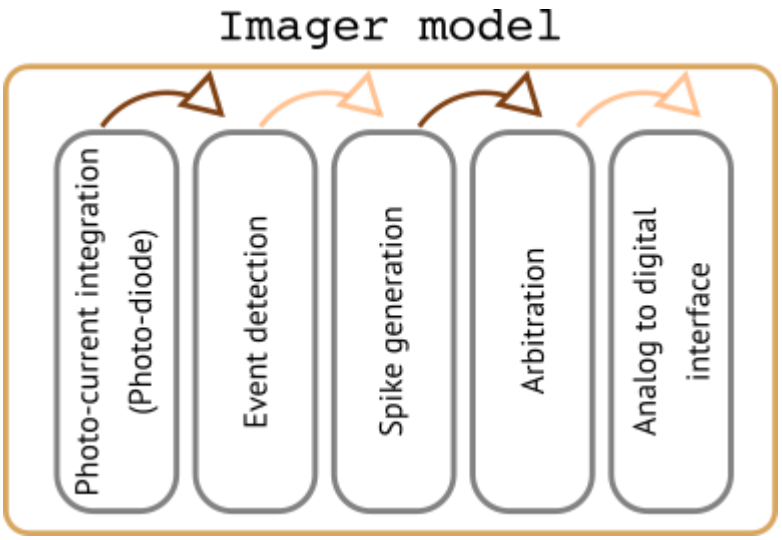


Figure 27 : imager model (Cubero et al. 2019)

Our imager model structure is presented in Figure 27. The image acquisition consists in 5 stages, typical of the state of the art. Depending on the imager type, one or more blocks can be removed or by-passed. For example, blocks like event detection, spike generation, and arbitration are typically related to specific types of neuromorphic sensors only. Next, we describe each of these stages:

Photo-current integration

This stage, from the code point of view, relates to the module dataload in Figure 28. We typically load a standard dataset that could include unknown pre-processing and compression functions. Ideally, this step should be done with non-preprocessed or “raw” images. Nevertheless, in this work we used 8-bit images from standard datasets as a first approximation.

In some cases, the image has to be pre-processed again from the dataset in order to make it compatible with the application scenario, typically by converting it from color space to gray space.

Event detection and Spike generation

Those stages relates to the module event in Figure 28. It encloses any functionality related to what we called before the “deep pre-processing”, i.e. any sort of pre-processing happening at pixel level or at matrix of pixels levels (not at column bottom or the periphery). One example is the Dynamic Vision Sensor (Posch et al. 2014), which detects (on pixel) local light intensity changes and sends an asynchronous signal (spikes) whenever this happens. Other kinds of pre-processing related to in pixel pre-processing, such as edge-detection (Yin, Chiu, and Hsieh 2016, 1), local-binary-patterns-detection (Gottardi and Lecca 2019), or Time-To-First-Spike (Guo, Qi, and Harris 2007) architectures are modeled here. The input is the loaded image, and the output is the pre-processed image in any encoded format related to the specific kind of pre-processing.

The reason why we separated the event detection from the spike generation was to prevent cases that an event occurring on one specific pixel does not trigger immediately a spike. Indeed, the spike can be triggered by the correspondence of several events of neighboring pixels such as the case of the local-binary-patterns (Gottardi and Lecca 2019) architecture. Moreover, one could imagine a general architecture that processes several events in neighboring pixels before sending any signal.

Arbitration

This stage relates to the module arbiter in Figure 28. It represents a specific functionality of asynchronous-spiking-imagers: asynchronous and event-based imagers sends spiking signals asynchronously. That means that a spike can be generated ideally at any time without being synchronized with an internal clock rate. As explained by (Posch et al. 2014), when a spike is sent, another circuit handles spike read-out, which normally consists in keeping information such as pixel address, event type and time-stamp. This circuit also deals with cases of several spikes coming at very similar times, for example, by establishing a priority order for reading each of them. The process just mention before is closely related to the known “hand-shake” protocol for asynchronous read-out. This functionality is included into this block. We decided to make a separated module for it, in order to simulate very specific phenomena happening in the arbiter if required: for instance, delays and data loss due to limitations in the arbiter circuitry (for instance, for simultaneous read-outs). As first approximation (and as we explain further in chapter 7), we modeled the arbiter as a perfect one, and we let this code architecture for being expanded more in detail in further works.

For classic (or frame-based) architecture, clearly the arbiter is not needed (as is the case for the event and spike generation stages in the model and respective modules). For those cases, the main imager model simply bypasses the input to the output in the code.

Analog to digital interface

This step is related to the module analog-to-digital in Figure 28. Notice that in classic imagers, this block would simply reflect the functionality of the analog to digital (ADC) converter. However, this block also takes into account any functionality resembling what we called previously the “shallow pre-processing”. Then, here we take into account models of pre-processing done right after the read-out (analog side) or right after the ADC (digital side). Several types of shallow pre-processing are already suggested in the state of the art. For instance, 1-bit-edges detection (Soell et al. 2016).

Before we go more into details for specific architectures, we explain how the simulation engine uses the imager model class in order to perform the behavioral simulation.

3.1.2. Imager model module

The simulation engine is in charge of creating an instance of a class “imager”, which is a sort of “skeleton” that creates instances of 5 different subclasses for assembling all the parts of the imager system. Notice that from now on, we refer to the “imager system” or to the “smart imager” as the stages from image acquisition to the shallow pre-processing. All the rest is outside the imager (in another layer for 3D IC technologies, or in another chip). Those 5 subclasses are contained in 5 different modules represented in Figure 28 for code modularity. Each module defines a specific task inside the imager[2], and each class inside each module represent a different variation (e.g. different kinds of pre-processing, image acquisition, etc). Moreover, the engine selects objects from which class to instantiate based on the configuration file provided by the user.

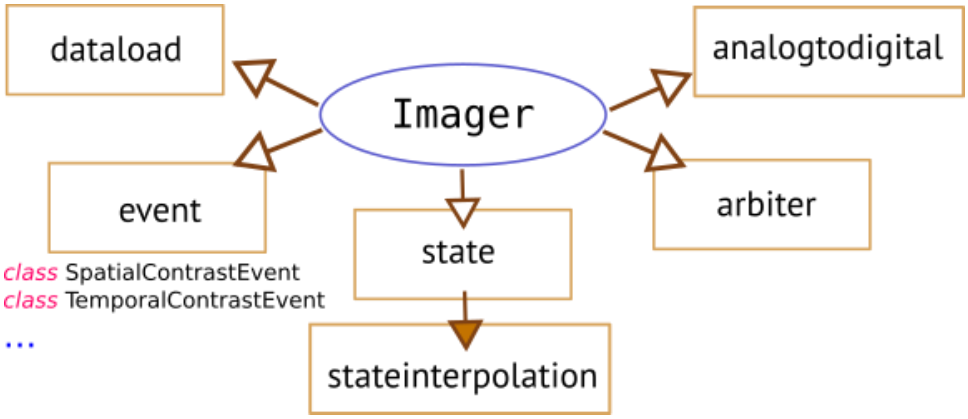


Figure 28 : example of the imager model implemented in Python modules (Cubero et al. 2019).

Once the imager model is ready to use, the simulation engine launches a simulation loop (further discussed in 3.1.3. Simulation engine loop). The point is that, during this loop, the imager simulates the behavior from image acquisition to imager system output. The input data for performing the simulation comes from a dataset, for which a specific data-loader class is automatically instantiated. Then, the output from this first stage is rendered (by a specific class-object in a secondary module and selected accordingly to the simulation settings). Indeed, there are two kinds of outputs generated (rendered) in this stage: one is a complete set of all outputs from this stage in a non-human-friendly format that is buffered in the hard-drive before the next stage. The other kind of rendering is for visualization purposes, so the user can see how the imager system output looks like and debug issues if required.

Once the simulation engine is finished, the main program launches the second stage: the processing. Notice that we call here processing everything that is after the imaging system. Similarly,

[2] We decided to make this separation between modules in order to isolate different phenomena while making code development easier. Moreover, the different modules are selected as they represent a good generalization of different imager types in the state of the art.

to the first stage, the processing follows the same kind of simulation loop, and has specific data-loaders and renderers. The difference is that here, we only focused on applying state of the art algorithms, without taking into account hardware constraints. For example, we used directly algorithms from the OpenCV library, and only wrapped them with the rest of the whole simulation pipeline. Once the processing is finished, its output is buffered again to save the intermediary steps (notice that this buffering becomes important when the size of all outputs and internal variables becomes greater that the RAM of the computer running the program). Since the output from this stage (in our cases) was more abstract or harder to visualize respect to last stage, we did not create an automatic visualization output for this stage. Notice that this stage could represent a CNN, a Random Forest, a Support Vector Machine, ROI proposal algorithms, among others.

One important aspect was that sometimes we required specific information about memory and runtime of the processing phase, in addition of the algorithm output. Moreover, sometimes specific sections of code were written in C++ language. For those cases, we wrapped the C++ code by executing it as a sub-process from Python. Moreover, we slightly modified the C++ code to output relevant information about memory and runtime, in addition of the main output. Moreover, we added an optional feature for executing compiled code in the processing stage in Figure 26. This optional stage (not present in the figure) was between the simulation engine and the processing block, and corresponded to a parsing task for solving any compatibility between our framework and external C++ code.

Once the processing is finished, its output becomes the input for the metrics (or third stage) part of the main program. Again, different types of metrics are defined for the different classes inside a module, and the program instantiates the right one depending on the simulation settings. One example of metrics that are evaluated could be classification accuracy, detection rate, recall, and others. The final output is a single file summarizing results as a Python dictionary. This file can be used latter for statistical data post-processing, and visualizations leading to final conclusions.

3.1.3. Simulation engine loop

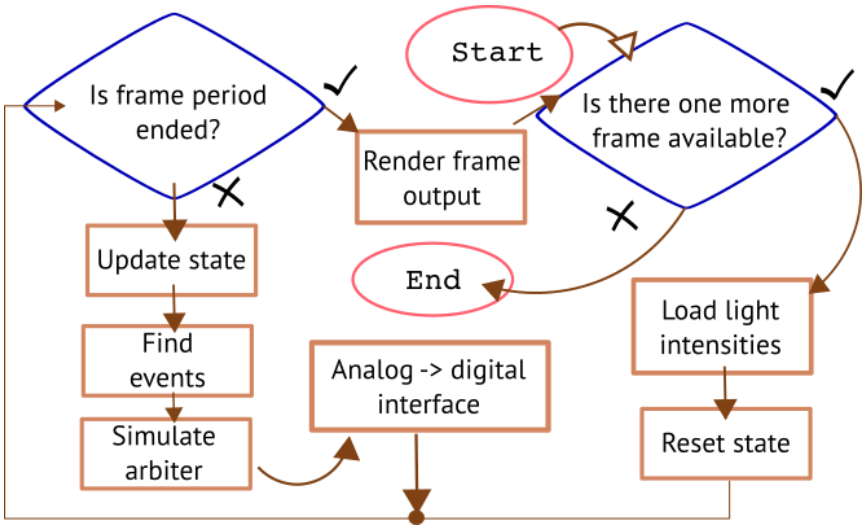


Figure 29 : simulation loop (Cubero et al. 2019).

The simulation engine applies a loop as shown in Figure 29. The simulation engine uses the imager object, which has been already setup following the user settings, in order to implement the loop. Firstly, a frame is loaded as a way of representing the photocurrent in the matrix of pixels. Then, the module state, which hold specific state variables such as the pixel equivalent voltage values, is reset. Then, the loop wonders if the frame period is ended. This question is important for simulations in which one tries to simulate more complicated phenomena. For instance, we could say that each pixel sense-node voltage o, is linearly increased by a time step (which is a small fraction of the frame period). This linear increase (or any increase represented in the state-interpolation module in Figure 28) happens during the update state phase. Then, the phase of find events starts. Continuing with our example, after a linear increase in pixels voltages, we could simulate a sort of complicated event that depends on values of neighboring events, and that somehow affects their values once it sends a spike. In such, or even more complicated cases depending on transient phenomena, this small step linear increasing during the step update is important. After events have been found for a specific small time step, the arbiter is launched, then the shallow pre-processing (the analog to digital interface). After several time increments, the reference time would be equal or higher than the frame period. Then, the simulation loops renders and output, and starts loading a new image. If there are no more images available, the simulation engine loop stops.

For the case of simpler, classic (frame-based) imagers, the linear increase is typically not required. Then, the loop works in a simpler manner: an image is loaded, then the pixels reads are reset. After, the engine launches blocks from update stage to analog to digital interface only once. After that, the loop moves on to the rendering frame output step.

3.2. Imager model examples

3.2.1. Ideal 8-bit-depth Sobel oriented-edges extractor

This model represents an ideal edge-extractor, in the sense that it assumes that all the pre-processing is carried out with numbers in floating point representation (32 to 64 bit-depth[3]). All pre-processing operations (which end with the edge-extractor output) are included in the shallow pre-processing module (see Figure 28). Again, depending on the user settings, the simulation engine selects the correct pre-processing class to use.

For this ideal case, the user selects the “8-bit Sobel edge extractor” imager-type. *Notice that the 8-bit part comes from the fact that input images for the simulation engine are typically codified in 8-bit for a given color space. Nevertheless, we do not simulate any effect due to electronics implementation in this model*. The simulation engine selects the right imager-class to use, and the imager class automatically instantiates the appropriate pre-processing class from the shallow pre-processing module (notice that several types of pre-processing are possible along with the same imager module, so this is also present in the user settings). We will go more in detail about the

[3] Due to specific aspects of the Python language and its packages, floating point numbers can be either represented in 32 or 64 bit-depth. In this work, we care about computations with low bit-depths (e.g. equal or lower than 8) respect to “high-resolution-bit-depths” (e.g. 32 bits or more). Then, we do not make a distinction in our analysis between 32 and 64 floating point representations in Python, and we refer to both in general as “floating point representation”.

mathematics and computations performed for the edge-extractor on chapter 4. Here, we specify how computations happen from a programmatic point of view.

The pre-processing class for the high-resolution edge-extractor contains a method that embeds all its stages: the function detect. It implements other class methods to simulate behaviorally the shallow pre-processing. The successive steps are:

1. Read and buffer the entire image
2. If binning is “True”, then apply 4x4 binning to the image (floating point). If binning is false, bypass the output from last step.
3. If apply blur is “True”, then select kernel based on user settings. The options are the kernel type (Gaussian, average) and size (given by a valid integer, e.g. 3, or 5). After that, apply the blur kernel to the entire image (floating point). If apply blur is false, bypass the output from last step.
4. Use the output from last step to obtain the gradient x component for each pixel. The gradient kernel type comes from the user settings (e.g. simple linear as the one used by (Dalal and Triggs 2005), logarithmic like in (Omid-Zohoor et al. 2018), and Sobel as the one used by (Soell et al. 2016)), in this particular case the Sobel is used. Operation happens in floating point, and the output are two matrices: one containing the absolute value (floating point) of the X component, and other containing the sign).
5. Do the same as step 4, but for the y component.
6. Use outputs from steps 4 and 5 to computes (floating point) the pixel gradient magnitude, and compare (floating point) the approximated gradient-magnitude with a threshold (provided in user settings). If the gradient magnitude is higher, then the edge magnitude (for that pixel) is the gradient magnitude itself (floating point). If the contrary occurs, the edge magnitude is zero (Notice that then, the edge magnitude is the result after comparing the gradient magnitude with a threshold). This step outputs a matrix of magnitudes, which are always either a floating point value higher than the threshold, or zero.
7. Use outputs from steps 4 and 5 to approximate the gradient orientations. The angle is approximated with the inverse tangent function in the gradient components (including the sign), and projected into the I and II quadrants. Notice that each gradient angle corresponds to one pixel-address, and the angle automatically is put to zero if the edge magnitude is zero. The output from this step is a matrix with angles (floating point) for every pixel.
8. The class method “detect” outputs two matrices, the edges and orientations.

Notice that computing steps mentioned before do not take into account any IC implementation, and they do not happen as they would in an integrated case. Nevertheless, steps mentioned before only try to assess how a particularly hypothetical pre-processing circuit, with high-bit-depth capability, compares with realistic architectures.

3.2.2. Analog linear oriented gradient extractor

In this case, we try to give a rough estimation of the output given by a more realistic edge-extractor implemented in analog electronics. In order to do so, we add noise to the signal before the output. This noise corresponds to a quantization noise related to a user defined (in the user settings) “equivalent number of bits” (ENOB). The computation sequence goes as following:

1. Steps 1 to 5 happen as for the high-resolution edge extractor (the simple linear kernel and an average kernel of size 3 are always used).
2. Apply quantization noise to the x gradient component.
3. Apply quantization noise to the y gradient component.
4. Use outputs from steps 2 and 3 to compute the edge magnitudes. In this case, a floating-point comparison between approximated gradient magnitudes and a user-defined threshold is the edge magnitude. That means that the edge magnitude takes a Boolean value (1-bit).
5. Use outputs from steps 2 and 3 to calculate the orientations. This time, floating-point values output from the inverse tangent are projected into 4 main directions: 0, 45, 90 and 135 degrees (or automatically zero if the edge magnitude is zero).
6. The returned values are two matrices, one for the edge magnitudes and other for the orientations.

In next sub-section we explain further how we apply the noise in steps 2 and 3.

The "apply quantization noise" function

Notice that this noise is supposed to represent the total noise coming from the analog pre-processing, as well as the temporal noise. That is, because it was easier to model for our framework. However, without a more detailed simulation we cannot know the spectral power function of the noise, so we are limited to choose arbitrarily an ENOB that seems reasonable. For design purposes, this ENOB states a first approximation of the noise budget for a particular pre-processing architecture. We try to take into account temporal and FPN sources mentioned by (El Gamal and Eltoukhy 2005) in a simplified form (e.g. by an equivalent quantization noise).

The apply quantization noise function takes as input a matrix representing a signal (gradient component along a specific axis for each pixel), and a value for the equivalent number of bits or ENOB. Its output is a quantized signal. The objective is to introduce quantization noise in order to simulate the impact of imperfections corresponding to analog electronics. We use the mid-thread quantization approach as described by (Wannamaker et al. 2000):

1. Calculate de number of quantization steps $nSteps = 2^{ENOB} - 1$ (mid thread)
2. Let $sMax = 255.0$, and $sMin = 0.0$
3. Calculate in floating point:

$$\delta = \frac{sMax - sMin}{nSteps}$$

Equation 2

4. Calculate the intermediary step qS (floating point):

$$qS = \left\lfloor \frac{1}{\delta} \cdot \boldsymbol{I} + \frac{1}{2} \right\rfloor$$

Equation 3

Where the brackets above represent the floor function (which spans for every single pixel), and $\boldsymbol{I}$ is the input image.

5. Finally, the output value is (floating point):

$$output = qS * \delta$$

Equation 4

3.2.3. Digital linear oriented gradient extractor

This model tries to emulate an edge extractor implemented in the digital domain. Then, we firstly apply the quantization noise before computing the pre-processing steps. The only function that happens before quantization is a 2x2 binning, since we assume that it happens directly in the matrix of pixels. The programmatic model is similar to the analog edge extractor. The differences are that the quantization is applied earlier, and the quantization function does not multiply by delta at the end. Indeed, the last multiplication by delta (a float) is unrequired, since the quantized output would be a matrix of integers. Later operations can be performed in integers only, which corresponds to digital operation with a fix bit-depth (as long as the maximum number is not attained).

The function detect goes as follows:

1. Buffer the 8-bit gray image.
2. Quantize the signal (image intensities) to a specific ENOB.
3. Apply blur and round output to integer values.
4. Obtain gradient (simple linear kernel) as integer operations.
5. Approximate gradient magnitudes by adding the two gradient components, and the edge magnitudes are calculated by comparing gradient magnitudes with the floating-point value of the threshold over delta (thus comparing an integer type with a float type).
6. Orientations are obtained with the inverse tangent applied on integer values and projected as for the analog edge extractor.

3.2.4. Logarithmic oriented gradients extractor

This model represents the logarithmic gradients as calculated by (Young et al. 2019), and related to Figure 5, Figure 6 and Figure 7. For this pre-processing type, we defined another attribute called qMinEdgeMag and set it to 1.0. Moreover, we changed the threshold attribute, whose value depends on the user settings, for the “ratio threshold”. In next paragraphs, we explain further those variables. Now, we specify the steps in the detect method for the logarithmic edge extractor class:

1. Buffer the entire 8-bit image.
2. Apply 4x4 binning (floating point) is requested by the user settings.
3. Apply 3x3 average blur if requested by user settings.
4. Gradient components are calculated as follows:
    a. The input of the function is a gray (potentially binned and blurred) image.
    b. For the x component, each pixel-gradient-value is the division (floating point) of the right neighbor-intensity-value over the left one. For the y component, the division is (in floating point) of the lower neighbor over the top one.
    c. Ratios (for either of the two components) are compared with the ratio-threshold attribute. If the ratio is higher or equal than it, then the component-magnitude is set to 1.0. If the ratio is lower than one over the ratio-threshold, then the component-magnitude is set to -1. Otherwise, the component-magnitude is set to 0.

5. Edge magnitudes are calculated by comparing the sum (in absolute value) of the two components with the attribute qMinEdgeMag. Recall that this attribute is 1.0 by default, meaning that the edge magnitude is 1.0 If either of the two components is not 0.
6. The orientations are also computed with the inverse tangent function, and projected to the I and II quadrants (taking sign into account).

3.2.5. Relative edge extractor

This shallow pre-processing class for the relative-edge-extractor defines an attribute called intensity-constant, which will be important during the edge magnitude calculation. The value for this attribute is set by the user settings. The function detect goes as follows:

1. Buffer the entire 8-bit image
2. Apply 4x4 binning (floating point) if requested.
3. Apply 3x3 average blur (floating point) if requested.
4. Get gradient components identically as for the case of the analog edge extractor.
5. Apply quantization noise with a particular ENOB to each component (again, identically to the analog edge-extractor).
6. Approximate edge magnitudes as follows:
    a. Take the gray image (potentially binned and/or blurred), and the two gradient components.
    b. Calculate the gradient magnitudes as the sum (in absolute value) of the two components. Then, calculate the edge magnitude by comparing the gradient magnitude with the gray image multiplied by the intensity-constant attribute. Every pixel address where the gradient is higher than the image times the constant, the edge magnitude is 1, and otherwise it is zero. All computations happen in floating point representation.
7. Orientations are calculated in the same way as for the analog edge-extractor.

3.2.6. Dynamic vision imagers

Dynamic Vision Imagers taken into account in (Posch et al. 2014), and related to Figure 15, typically have a high temporal resolution thanks to the in-pixel pre-processing (event detection). Such event corresponds to the local, and relative, light intensity change in time. Whenever such change is detected, a spike encoding the event is generated asynchronously. For modeling the behavior of this kind of imager, and in order to integrate it with our simulation framework, we observed the following: The pre-processing happens in-pixel, so this is reflected in the code by models appearing in the deep pre-processing modules (contrary to the shallow pre-processing module, typically used for edge extractors).

We found actually two ways of simulating the dynamic vision sensor behavior: one possibility was to take video frames (e.g. at 30 or 60 fps) and suppose that, for each frame, the intensity reading reflects a linearly-increasing-impinging-light on the pixel. That means that, at time 0, the light intensity is zero, and at time of read-out, the light intensity was at its final reading value. Then, for each frame, a shorter time step (close to the temporal resolution) would be used for updating light-intensity-change-reading for every pixel. Such update would assume a linear increasing of the total intensity as

mentioned before (by using the state interpolation module). Then, when an event is detected, an output is generated with the pixel address, the polarity (plus for increasing and minus for the opposite) and the time stamp. At the same time that this output goes, the pixel difference value is reset to zero. Nevertheless, the light intensity difference continues to grow after that until the frame period is reached.

The model mentioned in last paragraph presents one fundamental problem: it assumes that light intensity impinging the pixel follows a linear increasing whose values are determined by zero and the pixel value obtained from a frame based approach. Nevertheless, there is no physical reason to assume that it is the case. Indeed, the pixel value corresponds to all photons that had impinged the pixel during the integration time, which for an imager at 30 or 60 fps is much slower than the temporal resolution of a typical DVS (in the order to micro-seconds). Other attempts to simulate the DVS from frame based imagers could be to obtain a better interpolation of light intensity evolution along time, however, that could lead to even more complicated models that are still trying to recover fast phenomena from relatively small frequency image samplings.

To circumvent the issue mentioned in the paragraph above, we decided to synthesize a video dataset at a high frame rate, as further explained in chapter 7, so we could better approximate the instantaneous light intensity impinging at every pixel. Then, instead of using the state-interpolation module, we used an external 3D-scene simulator ("Blender" n.d.) for creating a synthetic high frame rate video, the frame rate being the temporal resolution of our simulated DVS. For this case, the simulation loop in Figure 29 runs from update state to analog-to-digital-interface blocks only once for every frame.

Continuing with the model for Dynamic Vision Sensors, we explain how events are detected. Indeed, even though the instantaneous light intensity is approximated only with the current simulation frame, the conditions of an event[4] happening depend on the history of previous events in the image. The reason is that each time that an event occurs at any pixel, its value for the cumulated intensity difference resets to zero. Moreover, for calculating the intensity change after any event, the pixel takes as reference value the instantaneous intensity at the moment of the last event. Thus, calculating lighting differences by simply subtracting two subsequent frames would be incorrect. The right way to do it, is by subtracting from the instantaneous intensity (from the current simulation frame) the values of light intensity at the last event for each pixel separately (the reference value of each pixel is different and may change asynchronously). For instance, in Figure 30 , light intensity evolves along time from left to right. We take a look at a pixel highlighted in orange in the matrix of pixels (blue rectangle). At time t_2, there are two possible ways of calculating the gradient: one gives a difference of |a| and another a difference of |b|. However, |b| is large enough to cause an event, whereas |a| is not. Then, the pixel detects an event at time t_2 only if the last event happened at time t_0 (or possibly before t_0), but not if the last event happened at t_1, which changes the reference lighting for computing the gradient at time t_2.

In order to take that into account, we defined a state variable that holds all reference intensities for the pixels, and another state variable representing the instantaneous intensity (obtained from the current simulation frame). Both variables can change for each event detected.

[4] Recall that an event corresponds to a light intensity change higher than a certain threshold, defined by the temporal contrast sensitivity.

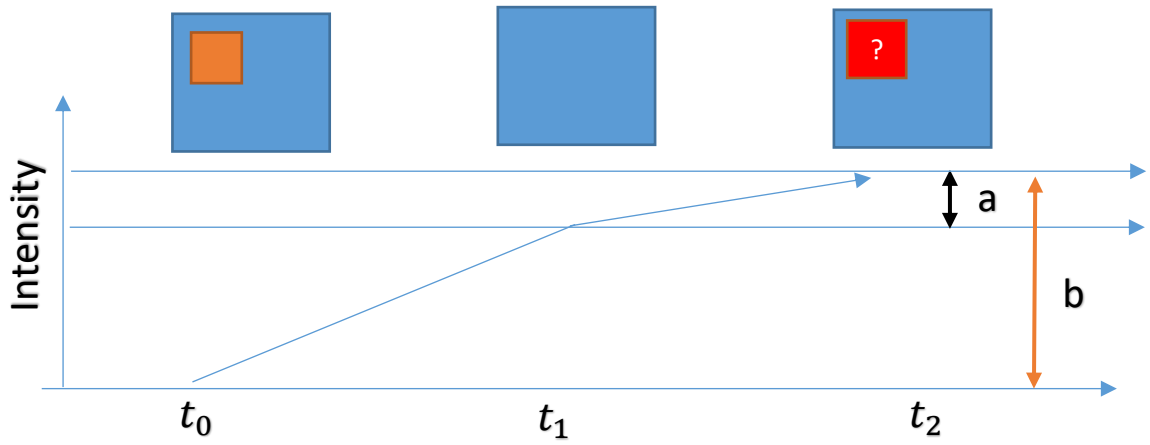


Figure 30 : example image of how an event at time step T_i can depend on a time step simulation T_{t-2}.

We took into account several aspects for synthesizing the high frame rate video. Firstly, the frame rate should be around or higher than 1000 frames/second. That would, at least, correspond to 1 ms of temporal resolution. Moreover, we used the Cycles ("Blender 2.93 Manual, Cycles" 2021) renderer engine, which seemed to better simulate photonic noise with respect to the standard Blender renderer. More details about the generated datasets will be given in chapter 6.

3.3. Conclusions of chapter 3

In this chapter we described the general models and code architecture for simulating different types of imagers. We specified our simulation pipeline in order to produce A.I. benchmarks in chapter 6. Moreover, we explained how we simulated behaviorally specific imager types, and how we approximated the impact of hardware constraints. Those specifications will be important when benchmarks are presented, since conclusions derived from those are limited by the fidelity of our models.

Chapter 4: Region proposals pipeline design

In this chapter, we study the feasibility of implementing Object Detection (OD), either in real time (e.g. at 30 fps or more) or not, on a dedicated near-sensor low power architecture (e.g. in the range of tens of pJ/pixel/frame). We analyze methods that are more classical and power-hungry such as approaches using desktop computers, or dedicated servers that can exploit powerful hardware, like a graphics card unit (GPU), for massive runtime accelerations. However, our near-sensor approach demands hardware-aware algorithms that consider both AI performance (e.g. classification accuracy, intersection over union, runtime, etc...) and circuit architecture performance (power, speed, silicon surface…). This often leads to an iterative algorithm-and-hardware development, in which changing one of them may affect the other. For instance, if an algorithm relies on 32-bit division operations, but the hardware can only support 8-bit summations, then one has to adapt the algorithm to a lower bit resolution and simpler mathematical operations, or to increase-complexity / adapt the integrated electronics, or to find an intermediary trade-off by changing both of them.

The state of the art shows progress in implementing OD in embedded architectures. Some examples like (J.-H. Kim et al. 2019) (Suleiman and Sze 2014) (Choi et al. 2014) (Omid-Zohoor et al. 2018)*, among others, start from a stable algorithm running on a complex hardware (e.g. a modern desktop pc), and then try to adapt it to low power-integrated-architectures. Most of the works we found rely on frame-based approaches, since frame-based (or just typical) cameras are user friendly and easily supplied. Such approaches typically focus on solving A.I. computer vision problems by using single-static images, one by one.*

In that context, this chapter aims to explore another possible pipeline and architecture for embedded OD, instead of the sliding window, namely, the usage of Regions of interest or Region proposals methods to reduce time and power complexity. First we contrast ROI based OD algorithm with the more classical sliding window approach. Then, we go deeper in selecting and understanding one specific ROI proposal algorithm potentially suitable for embedded integrated electronics: Edge-Boxes. Next, we discuss how to generate the input for the selected algorithm: the oriented gradients map. Finally, we present a preprocessing pipeline for computing on the fly oriented gradients for low power applications.

4.1. ROI proposals vs. typical sliding window approaches

Frames have standardized representations on different color spaces: one can imagine a frame (or simply, an image) as a 3D matrix. When seen from above, this 3D matrix looks like a 2D matrix and each "entry" is a pixel. Then, if we turn around a little bit to the side, we will be able to see the third matrix dimension (or its depth). This dimension (normally consisting in 1 to 4 values) are the pixel values for a specific color encoding (e.g. RGB, gray, etc…). One can notice that each image (or frame) only contains static information, and if one wants to recover dynamic information, such as the speed at which objects (or borders, points, etc…) are moving, then one would need at least two frames.

OD runs successfully on frames, and has interesting applications when information about different objects in an imaged scene is important. More specifically, if one wants to know the location (in a projected 2D image space) of several objects, and the nature (class) of each one as well. This process is carried out with different main strategies in the state of the art. For instance, the simplest is to train a classifier and "test it" as exhaustively as possible along the frame for finding objects (sliding window) as made by (Dalal and Triggs 2005). Another option is first to generate potential regions with objects in the scene with a lightweight algorithm (ROI proposals algorithm) and then to pass those regions to a classifier[5], as done for example by (Girshick 2015). This last example matches the Fast R-CNN architecture (Girshick 2015). Continuing with the improved version of Fast R-CNN as example: Faster R-CNN (Ren et al. 2017), one of the main ideas is to change the ROI proposals algorithm for a CNN that is in charge of detecting such regions, and then using ROI pooling[6] in the CNN-feature-extractor output for passing selected regions to the classifier. Finally, one pipeline (that is the least relevant for this work) is to remove the regions proposals section completely, but without having to search exhaustively for every possible position and scale in the image. That is, by computing several regions (ideally "at the same time") and along a less dense grid in the image. Two known examples are the single shot detector (Liu et al. 2016) (SSD) and You Only Look Once (Redmon et al. 2016) (YOLO) CNN architectures.

In Figure 31 we summarize some examples in the literature that we found closely related to our work, and we present (in red) the processing pipeline (already mentioned in the SoA) that we found interesting to implement in an embedded smart imager context.

[5] Notice that those regions are projected from an intermediate feature space if ROI pooling is used (Ren et al. 2017).

[6] ROI pooling is a way of using region proposals in a CNN-features-output and not directly in an input image (Ren et al. 2017).

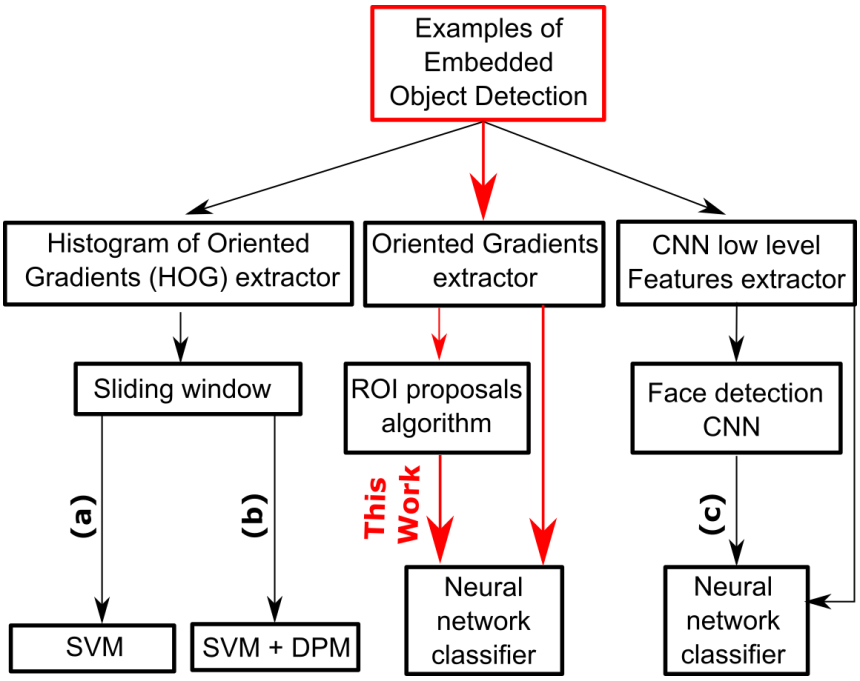


Figure 31 : Example of diagrams of pipelines for embedded integrated devices.

Figure 31 shows three different examples of OD in embedded devices. The red arrows show the pipeline of interest for this work. Figure 31 (a) corresponds to a sliding window approach with a support vector machine (Suleiman and Sze 2014). Figure 31 (b) is a similar case to (a), but the classifier is improved by adding the deformable parts model (DPM) (Omid-Zohoor et al. 2018). Figure 31 (c) corresponds to an identity-verification system with a face detection network, and a subsequent face classification network for verifying user identity (J.-H. Kim et al. 2019).

As mentioned in last paragraph, Figure 31 (a) and (b) show one possible approach for OD: the sliding window. In order to explain it, we can start by hypothesizing simpler case scenarios: we can assume that a target object will appear on the image with a specific size and aspect ratio. That is, if one imagines a rectangle perfectly englobing the object[7]. For example, we can say such rectangle size is 15 pixels high by 10 pixels wide. Then, one could train a classifier to label images of size 15 x 10, and with the object of interest nicely centered on this small image. Finally, if we want to locate and classify any object in a wider test image (for example, with size of 100 x 300 pixels), we can evaluate sub-portions in this test image, by taking groups of pixels of size height x width (15 x 10 for our example). We can do that with every possible group of pixels in the test image, with the condition that its size is 15 x 10, and then let the classifier to determine if such portion contains the desired object or not. This method is the so-called *Sliding window approach,* used for example by (Omid-Zohoor et al. 2018) (Dalal and Triggs 2005) (Suleiman and Sze 2014), and we can imagine it as a small rectangle of fixed size sliding along a wider test image. Nevertheless, the amount of tests (e.g. when placing the small rectangle on the test image for a test with the classifier) increases as $O(n^2 = height * width)$ with the test image size. Indeed, this situation gets worse when there are several objects of different nature to find, since the number of tests increases linearly (assuming one test, and model, per class) with that quantity as well. Moreover, the fixed-object-size assumption (15 x 10 in our last example) is not valid for many real scenarios. To circumvent that issue, works like (Omid-Zohoor et al. 2018; Dalal and Triggs 2005; Suleiman and Sze 2014) have implemented the so-called "*Image scale pyramid*" ("Pyramid

[7] i.e. This rectangular zone perfectly encloses the object limits (rightmost, leftmost, top and bottom) in the image.

(Image Processing)" 2021; "Scale Space" 2021): it consists on several sequential de-noising steps (for example, with a Gaussian kernel) and down-sampling one already blurred image by a factor of two along each height and width dimension. This process repeats several times, thus generating an image scale pyramid, where each scale represents a different pair of blurring and down sampling ("Scale Space" 2021; "Pyramid (Image Processing)" 2021). Then, the sliding window is applied exhaustively along this scale pyramid, which means that the classifier-testing time complexity increases linearly with the amount of scales as well. In addition, there is an overhead due to the scale pyramid generation itself. Those aforementioned factors can make the *OD* pipeline too expensive in terms of time and power for embedded devices targeting, for example, mega-pixel images as input and under challenging scenarios.

Another popular example for embedded applications, showed in Figure 31 (c), is face recognition and identity verification in mobile phones, like in (J.-H. Kim et al. 2019): firstly, the phone "has to" determine where in the whole image the face is, or if there is a face at all (which is analogous to the localization phase). However, this localization phase is not « class agnostic », since it is not scoring any object presence, but rather a specific target class. In this case, the ROI localization is a low power and weaker classifier, computed exhaustively along the image, and which is not capable itself of determining if a detected face corresponds to the registered-user or not. Secondly, it has to assess if the detected face corresponds to the registered user or not (with that being the classification phase).

The sliding window approach is simple and intuitive but the required computational complexity increases for high-resolution images. To circumvent that, the Region proposals or ***ROIs*** approach used for example by (Girshick 2015; Ren et al. 2017) could have advantages if the overhead caused by the ROIs calculation is acceptable. The principle is that, instead of "testing densely or exhaustively" the classifier, a previous algorithm is in charge of generating a certain amount of regions that are most likely to contain objects. The first potential advantage is that the amount of tested ROIs passed to the « energy hungry » classifier decreases enormously. The second advantage is that there is time saved when processing each frame since (ideally) only the relevant regions are tested. Those two reasons are critic in low power applications with objects of many different sizes. That is why we chose to explore a pipeline such as the one exalted with red arrows in Figure 31.

4.2. Selecting a ROI detection algorithm

There are several ROIs algorithms in the literature targeting non-constrained devices. One example is the Fast R-CNN (Girshick 2015). Afterwards it was improved and named Faster-R-CNN (Ren et al. 2017), and then Mask R-CNN (He et al. 2018) (the latter includes segmentation as well). Nevertheless, their integrated circuit (IC) implementation remains difficult. One example in the SoA (J.-H. Kim et al. 2019) has already tried this idea along with a hybrid (analog / digital) Convolutional Neural Network CNN (which resembles to an "embedded version of Faster R-CNN"), and for face detection/identity verification on mobile devices.

This aforementioned idea, from (J.-H. Kim et al. 2019), was to use a low power CNN (they implemented it in the analog domain, i.e. before the analog to digital converter) whose input are pixel readings, while its output is (or allows to obtain easily) regions of interest. Their processing pipeline is showed in Figure 32:

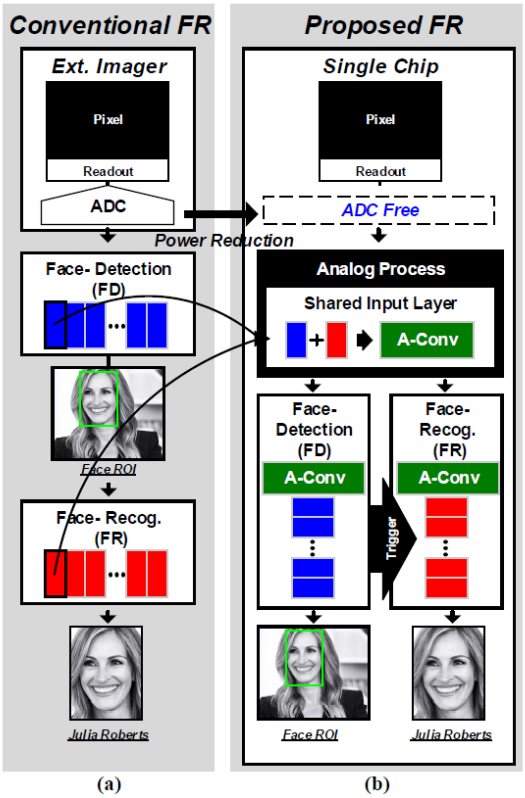


Figure 32 : "(a) Conventional face recognition system (b) Proposed face recognition system" (J.-H. Kim et al. 2019).

In their case, the regions of interest where related to potential faces, and when one region was detected, another CNN (on the digital side) was triggered for performing the classification (identity verification) phase. This approach is interesting, yet the justification for an analog CNN, complex to design and implement, is not completely clear. For example, this analog CNN has to be simple enough (i.e. just one convolutional layer in their work (J.-H. Kim et al. 2019)), and is aggressively quantized (3-bit weights in their work). Nevertheless, in their results, they report a very similar accuracy for the Hybrid CNN (for face detection) with respect to the usage of a Haar-Features (see Table 2). Then, this made us wonder if such aggressive quantization actually makes a CNN implementation near the ADC more interesting that a non-trainable approach. For instance, for implementing a trainable kernel (or one CNN layer) their design had to take care of an analog unit capable of multiply-accumulation operations. In addition, non-volatile memory reads for weights were required. Finally, the design had to include analog implementations of the activation function: ReLu. That leads to trade-offs typical of an analog design, such as fixed pattern noise, temperature-voltage variations, and non-negligible noise sources introduced by all added transistors and capacitances (which can accumulate after each operation).

Table 2 : The hybrid-CNN implementation versus previous works (J.-H. Kim et al. 2019).

	JSSC'17 [4]	JSSC'18 [5]	This Work
Technology	TSMC 40nm	Samsung 65nm	Samsung 65nm
Algorithm	FD: Haar-like FR: PCA+SVM	FD: Haar-like FR: CNN	*FD&FR : Analog-digital hybrid CNN*
Accuracy	81% @ 32-class in LFW	97% @ whole LFW	*96.18% @ whole LFW*
Resolution	HD	QVGA	QVGA
Power	23mW	0.62mW	*0.6198mW*

Based on factors mentioned previously, we decided to explore another approach (which resembles to an "embedded version of fast R-CNN or R-CNN"): instead of a simple CNN in the analog domain, we prefer to generate hand-crafted features that could be implemented on the sensor silicon die, like a gray-light-intensity gradient computation. An example is offered by (Omid-Zohoor et al. 2018): they showed a near sensor architecture that can improve dynamic range issues thanks to an approximation of the gradient as a ratio[8] instead of as a light intensity subtraction (or subtraction and addition, depending on the gradient kernel). Their pipeline is showed in Figure 33, where they introduce the ratio approximation by means of their "ratio-to-digital-converter" (RDC) (Omid-Zohoor et al. 2018).

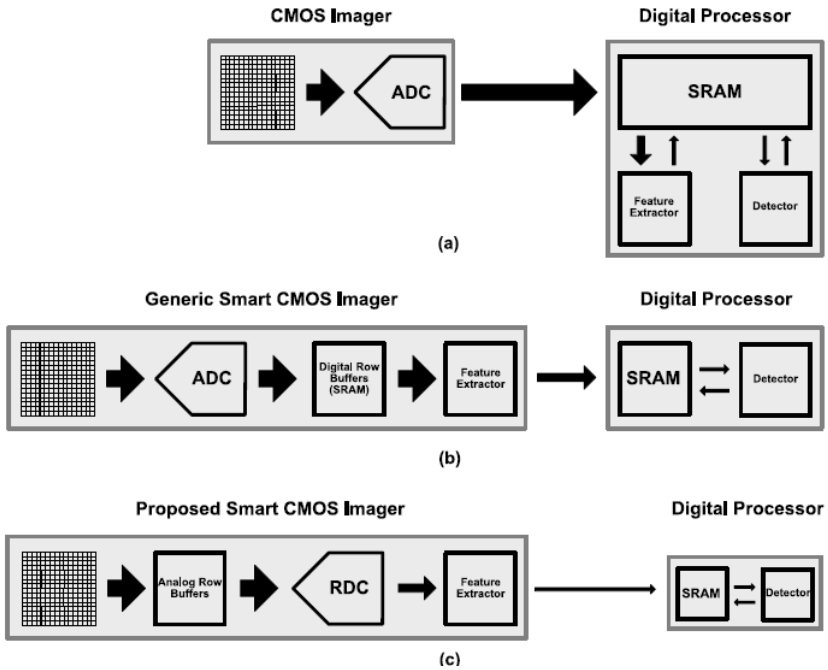


Figure 33 : (a) OD in general. (b) OD with embedded and digital pre-processing. (c) Their innovative approach with embedded digital and analog pre-processing. (Omid-Zohoor et al. 2018)

However, one question raises from the paragraph above: *how to use hand-crafted features for region proposals generation?* Indeed, in our state-of-the-art compilation, we found several algorithms that do not rely on CNNs for ROIs detection. Of course, such generated ROIs are good-candidate inputs for a potential CNN classification (or other type of classifier). Among those ROIs

[8] They (Omid-Zohoor et al. 2018; Young et al. 2019) called such approach the logarithmic gradients, since the gradient representation as a ratio aims to approximate a logarithmic analog to digital conversion which is less sensitive to local light intensity variations.

proposals algorithms, we selected Edge-Boxes (Zitnick and Dollár 2014). The reasons are, firstly, because its first stage depends upon an image of edges which could be feasible to extract with a low power embedded architecture, as done for example by (Choi et al. 2014). Secondly, because chapter 6 will show that it has reasonably good performance even for gray images. From Table 3, we observe Edge-Boxes has one of the lowest runtimes with good recall with respect to other approaches.

In order to avoid any confusion, in this work we call edges individual pixels with high local-intensity-gradient-magnitude. In the other hand, segments are groups of edges that are "related" by a clustering algorithm. Finally, contours, are groups of segments that form a more complex topology.

Table 3 : benchmark of performances for different region proposals algorithms: "… for IoU threshold of 0.7. Methods are sorted by increasing Area Under the Curve (AUC). Additional metrics include the number of proposals needed to achieve 25%, 50% and 75% recall and the maximum recall using 5000 boxes…" (Zitnick and Dollár 2014).

	AUC	N@25%	N@50%	N@75%	Recall	Time
BING [11]	.20	292	–	–	29%	**.2s**
Rantalankila [10]	.23	184	584	–	68%	10s
Objectness [4]	.27	27	–	–	39%	3s
Rand. Prim's [8]	.35	42	349	3023	80%	1s
Rahtu [7]	.37	29	307	–	70%	3s
Selective Search [5]	.40	28	199	1434	**87%**	10s
CPMC [6]	.41	15	111	–	65%	250s
Edge boxes 70	**.46**	**12**	**108**	**800**	**87%**	**.25s**

Introducing Edge-Boxes into the embedded OD pipeline raises the question of *"how and where to implement it"* (e.g. analog or digital, the exactly same algorithm of a modified version, etc…). Notice that Edge-Boxes digital implementation is not the objective of this work, but rather to assess if an imager with specialized pre-processing for it would be worth. For that purpose, we decorticate it in different stages, and try to estimate its complexity and associated speed in order to assess a potential digital implementation. Moreover, we try to assess if the modifications targeting a low power architecture do not degrade significantly the whole algorithm performance.

In the next section, we go more into details about Edge-Boxes: we decorticate and compare it with other approaches.

4.2.1. EdgeBoxes decortication

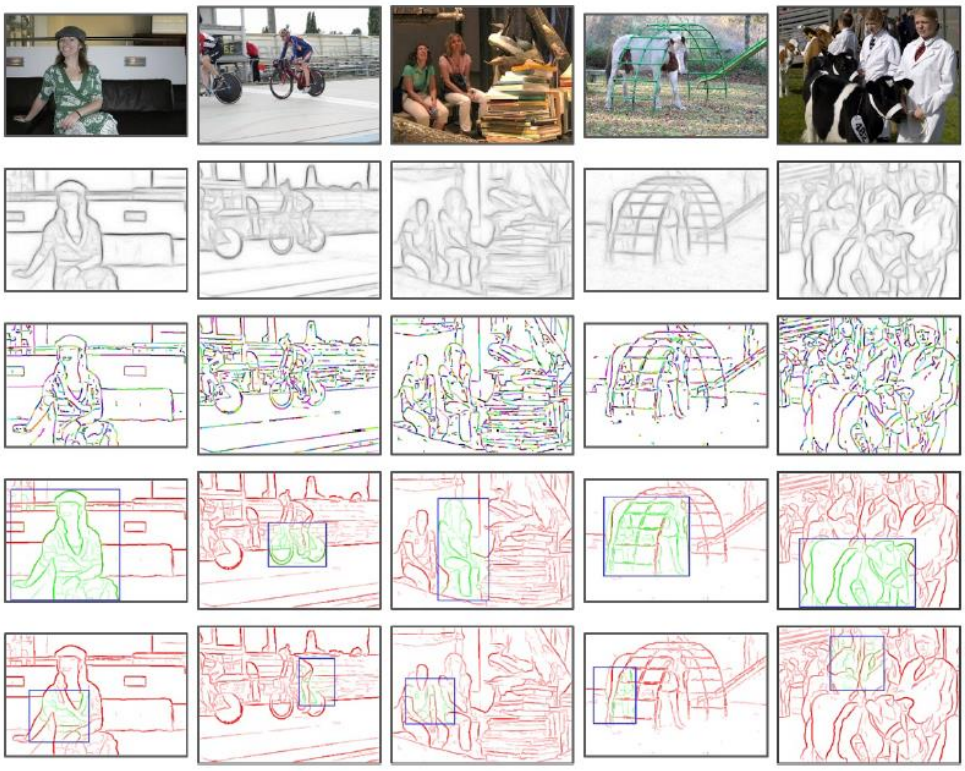


Figure 34 : principle of functioning of Edge-Boxes (Zitnick and Dollár 2014).

EdgeBoxes (Zitnick and Dollár 2014) is a region proposal algorithm: it takes an edges-map as input, and its output is a list of Region Proposals (ROIs). For instance, in Figure 34 (Zitnick and Dollár 2014) they present their stages from input to output. The input is a colored image, from which the oriented edges are extracted. Then, the algorithm tries to relate pixels into clusters (connected segments and contours), and then uses them for scoring a series of hypothetical boxes distributed along the image. Finally, it outputs those that are most likely to contain an object. The input edges map corresponds to two 2D matrices of equal size: one filled with values between 0 and 1 (**Mag**), and another filled with values between 0 and π (**Or**). Typically, the edges map derives from a standard image frame. In addition, both **Mag** and **Or** matrices-size (when projected into a 2D image[9]) are equal or smaller than the original image used to obtain the edges map[10]. When rendered, the edges map appears to "follow" the original image contours, and the more contrast is present along the contour, the brighter it will be. Indeed, a value for **Mag** at the position *ij* can be interpreted as the local and spatial light-intensity-gradient magnitude. The corresponding value in **Or**, at the same *ij* position, represents the local and spatial gradient orientation (in a 2D image space). Values in **Or** go between 0 and π and not to 2π, since two orientations pointing along the same straight line are considered as equivalent, even if they go in opposite directions. For instance, if a gradient orientation points towards $5\pi/4\ rad$, then its

[9] Notice that **Mag** and **Or** are a 2D matrices both of the same size (height x width), while the input image can have a more complex color encoding representation. Then, the original image size is (height x width x color-encoding-depth).

[10] The image size can be reduced due to border effects, or after down-sampling. Another case may be to up-sample an image before extracting its contour map, but this is not relevant for the present work.

corresponding value in the **Or-**matrix will be $\pi/4\ rad$. Regarding EdgeBoxes output, it is a list of ROIs. Each ROI corresponds to a 4-value-array containing information about the size and position (in image space) of a rectangle englobing a potential object. Finally, a fifth optional value for each ROI represents its score , which goes between 0 and 1, and estimates how likely an ROI actually contains an object (Zitnick and Dollár 2014).

In the next sections, we will go into more details about how to generate the input edges map (both for the general and for the embedded case scenarios). Until then, we will assume it is available for EdgeBoxes, and in order to decorticate it.

We base our explanation from both their publication (Zitnick and Dollár 2014) and source code (Dollar and Zitnick 2015b), since several details were not completely clear for us directly from their paper: EdgeBoxes is composed of 3 main pipelined stages: edge clustering, data structure preparation, and boxes scoring. We will explain them briefly as follows:

Edge clustering

During edges clustering, each *ij* value-pair conceptually corresponds to an "edge" if $\boldsymbol{Mag}(ij)$ is above a user defined threshold[11]. Besides, the algorithm tries to "merge" groups of edges into clusters based on a specific strategy. Clustering is important for computational efficiency at later stages. Regarding the merging criterion, it is as follows: starting from an edge *ij,* supposedly the only element on cluster *C*, the algorithm finds the 8-connected pixel to *ij*, namely $i + x, j + y$ such as[12]:

$$\boldsymbol{Mag}(i + x, j + y) > threshold$$

Equation 5

And,

$$x, y = \underset{x,y \in \{-1,0,1\}}{\mathbf{argmin}} |\mathbf{Or}(ij) - \mathbf{Or}(i + x, j + y)|$$

Equation 6

The "argmin" function implies that the algorithm is "looking" for the 8-connected edge whose orientation is the closest to *ij* 's. Of course, the case in which $x = y = 0$ is not relevant. Once a new edge is "added to the cluster"[12], $i + x, j + y$ takes the role of the new starting point for a next clustering iteration. This process stops when the cumulated orientation differences for C is higher than another threshold, or when there are not 8-connected edges[13] to the current *ij-pair*. If a clustering process stops for a cluster C, then the algorithm continues to "scan" along the contour map "searching" for individual edges that can start a new cluster (i.e. that have 8-connected edges, and which do not belong to any cluster yet). In addition, after clustering, the algorithm tries to eliminate clusters whose magnitudes are smaller than a user defined threshold. However, clusters with magnitudes lower than this threshold can be merged with other clusters if possible.

[11] Not to confuse with other thresholds, especially those that could have been previously used to generate the edge map.

[12] This is tracked by a matrix of id numbers, where each id maps to a specific cluster.

[13] Here, we refer to 8-connected edges as edges that are only one pixel apart, no matter in which direction (including diagonals). This is a distinction to other types of connectivity such as 4-connected edges that does not take into account diagonals ("Pixel Connectivity" 2019).

Once the clustering is done, the affinity parameter "a" is introduced to estimate how likely two segments (or edge clusters), whose "heads and tails" are not 8-connected, but that are "near enough"[14] from each other, can be deemed to "belong"[15] to the same contour. The formula for a is given below:

$$a(s_i, s_j) = \left|\cos(\theta_i - \theta_{ij})\cos(\theta_j - \theta_{ij})\right|^{\gamma}$$

Equation 7: (Zitnick and Dollár 2014)

In Equation 7, the affinity a of segments s_i and s_j is calculated as a function of a user defined parameter γ. Angles θ_i and θ_j are the overall angles for each segment, and θ_{ij} is the angle between the two segments. Another important aspect is that if segments are more than 2-pixel radius away, they get an affinity of zero. One example is illustrated in Figure 35.

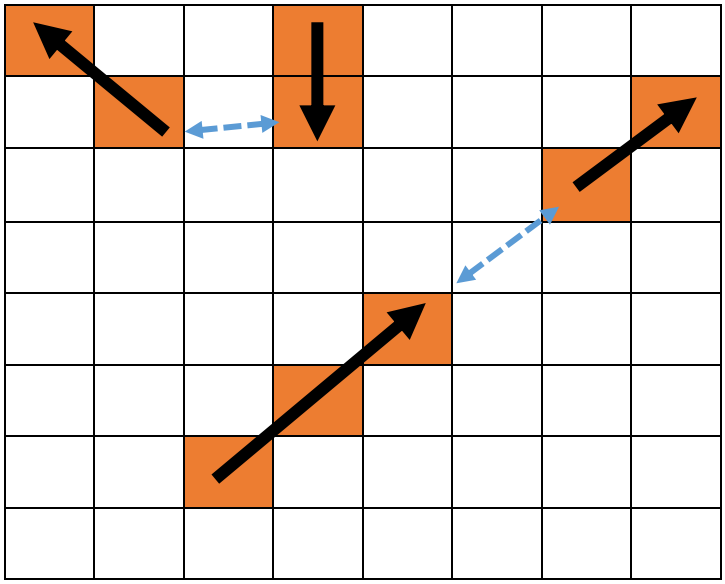


Figure 35 : example of how Edge-Boxes relates disconnected segments.

Figure 35 shows an example of how Edge-Boxes relates disconnected segments throughout the affinity variable. Orange squares represent pixels detected as Edges. The black arrows represent segments average magnitude and direction. Finally, the blue, discontinued and smaller arrows represent the affinity variable. Notice that segments on the lower right have higher affinity than the two in the upper left, due to the similarity of average angle. Also, segments separated by two or more pixels have no affinity between them.

During the clustering process, the algorithm creates new data structures (2D matrices) for holding relevant cluster information, such as cluster id, cluster average orientation and cluster average magnitude. In order to store and to further use the affinity, efficient data of non-fixed length are defined. Firstly, two vector variables hold the bottom-right edge coordinates for each cluster (the index corresponds to the cluster id). Secondly, a data structure, which is not a 2D matrix but rather "a list of

[14] Inside a radius of 2x2 pixels in the Author's source code (Dollar and Zitnick 2015b).

[15] From our perspective, the notion of « belonging to a contour » is subjective or hard to define. We simplify this by imaging that two segments are indeed forming a contour if it is very likely that a human- being would state it as true, and by means of simple visual inspection.

lists"[16], is the *segments Affinities (*represented as *_segAff and _segAffIds*[17] in the author's source code (Dollar and Zitnick 2015b)). Being those more abstract (complex) than the others, we dedicate the next paragraph to explain them.

In order to know which pair of segments (clusters) relate to any entry in _segAff, another variable (_segAffIds) becomes essential. Both _segAff and _segAffIds have the same shape. Moreover, since they both are lists of lists, any value (entry) corresponds to an index pair, similarly to a 2D matrix. For the case of _segAffIds, the first index *x* represents a particular cluster id or C_x (thus the amount of lists equals the number of clusters). The second index *y* iterates along a sub-list of all clusters that have a non-zero affinity with C_x. Notice that $y \neq$ segment id, and if one wants to obtain the second segment id (the first id is *x*) for a value pair (x, y) in _segAff, this value lies on the variable _segAffIds at the same index pair (x, y).

Aiming at a digital implementation, we enumerated the class variables created during this step and estimated its memory size:

[16] This distinction is important, since each « sub-list » within the list of lists, as well as the amount of sub-lists can grow in size in runtime (as opposed to a fixed size image). In addition, different sub-lists are not restricted to have the same length.

[17] Typically, the underscore before the variable name is a convention to indicate that it is a C++ class variable. This is not relevant for understanding this work, yet we clarify it to avoid confusion.

Table 4 : most-relevant-variables size-estimation for the first EdgeBoxes stage; segments = $2^{int} = 2^{16}$ (worst-case).

Name	*Native type*	*Derived type*	*Size Formula*	*Max size (kBytes)*
_segIds	int	Array	int*h*w	500
_segMag	float	std :: vector	float*segments	262,144
_segAff	float	std :: vector	float*segments*totalNeighbors*2*2	25165,824
_segAffIdx	int	std :: vector	float*segments*totalNeighbors*2*2	12582,912
_segC	int	std :: vector	int*segments	131,072
_segR	int	std :: vector	int*segments	131,072
map	int	vector	int*segments	131,072
meanX	float	vector	float*segments	262,144
meanY	float	vector	float*segments	262,144
meanOx	float	vector	float*segments	262,144
meanOy	float	vector	float*segments	262,144
vs	float	vector	segments*float	262,144
cs	int	vector	segments*int	131,072
rs	int	vector	segments*int	131,072

In Table 4, we estimate the size of most "memory hungry" variables generated in the clustering stage. Values for int and float are 16 and 32 respectively, the maximum amount of segments, or simply segments is $2^{16} = 65536$, and totalNeighbors $= 24 = (2 * radius + 1)^2 - 1$, with radius = 2. Notice the size formula for both _segAff and _segAffIdx is multiplied by 2 two times; the first factor takes into account that any affinity is copied twice (for example, the affinity for segments C_x and C_y will appear in both _segAff[x] and _segAff[y] lists). The second factor takes into account that neighbors can be at either at the head or at the tail of any segment. One can immediately notice the high memory values (for instance, more than 25 Mbytes for _segAff) required. As we will show later, this example is illustrative, but it lacks accuracy for selecting a realistic amount of segments. As we did not develop a theoretical formula for a better amount-of-segments estimation, we decided to obtain it experimentally as it will be shown later in section "4.2.2. Preliminary memory estimation".

Once finished all those initial steps, the algorithm continues to the rest of data structures preparation.

Data structures preparation

This phase aims to maximize the computational efficiency for determining the "objectness" (Zitnick and Dollár 2014; Alexe, Deselaers, and Ferrari 2012) of ROIs. From here, we find useful to introduce already how Edge-Boxes estimates this objectness: the idea is to estimate it by "counting" the amount of connected contours that trespass, or overlap, with the ROI borders (again, considering the ROI as a rectangle). In other words, the algorithm considers that if an object is located inside any hypothetical ROI, then most of the existing contours inside it would not trespass the ROI limits. However, when applied, this concept brings several complications: For instance, several clusters related by their corresponding affinities can form a contour, and then, a contour can appear to be "disconnected". Moreover, another question is how to handle contours that could have several branches. One final example is the time complexity of searching contours along ROI borders when estimating its objectness (box scoring): as discussed later in the profiling, this operation (box scoring) happens many times during Edge-Boxes execution.

Complications such as those mentioned above bring the importance of data structures preparation. During this stage, several variables are created to increase the estimation speed for ROI objectness (or, box scoring, which is the next stage). We do not enter into details of each variable's objective. Indeed, this is not necessary in order to have a first approximation of the memory usage, and it could make the present writing too tedious[18]. We repeated the same steps as before in order to obtain estimated variable sizes depicted in Table 5.

Table 5 : most-relevant-variables size-estimation for the second Edge-Boxes stage; h = w = 500 pixels.

Name	*Native type*	*Derived type*	*Size Formula*	*Max size (kBytes)*
_scaleNorm	float	std :: vector	1000*float	40
_segIImg	float	Array	(h+1)*(w+1)*float	1004,004
_magIImg	float	Array	(h+1)*(w+1)*float	1004,004
_hIdxs	int	std :: vector	h*int	1
_hIdxImg	int	Array	h*w*int	500
_vIdxs	int	std :: vector	w*int	1
E1	*float*	*Array*	*float*h*w*	*1000*

From Table 4 and Table 5, one can observe that variables in the first one depend on the amount to clusters or segments. On the other hand, the latter table presents space-complexities depending only on image dimensions. Once this stage is completed, the algorithm move on to the box-scoring stage.

Box scoring

In this phase, Edge-Boxes applies the same idea of the sliding window approach. However, the benefit respect to a sliding window without Edge-Boxes, is that objectness calculation is expected to be less "expensive". Then, it can be densely (not necessary exhaustively) estimated along the image. Edge-Boxes searches for ROIs almost exhaustively in the image: it "distributes" a series of hypothetical ROIs along it. Those hypothetical ROIs correspond to groups with different sizes and aspect ratios[19]. User parameters control boxes density (spacing), aspect ratios and sizes.

During this stage, the algorithm "slides" across all hypothetical boxes and scores them. For the scoring, Edge-Boxes adds borders completely inside the ROI, and the amount added depends on cluster magnitudes. In addition, it penalizes when clusters intersect with the ROI borders, or when clusters inside the ROI are likely to form a connected contour with clusters outside. In order to figure that out, the affinities variables contain the required information to link groups of clusters. Notice that these variables relate to clusters that could form a completely connected contour. In addition, even though the scoring happens over a potentially big amount of boxes, not all of them are in the output. The user can select the desired amount of ROIs, and Edge-Boxes will provide the top ones accordingly. Of course,

[18] Those variables are further explained in the EdgeBoxes paper (Zitnick and Dollár 2014).

[19] Actually, the process is more complicated than that, in the sense that after scoring a box, if a score is high enough, then Edge-Boxes tries to « improve » that score by testing other boxes with the same size and aspect ratio in the surroundings. This idea is called « box refining » by the author (Zitnick and Dollár 2014).

in order to avoid redundant boxes in the output, a non-maximum-suppression[20] step becomes important (which is already done in the original algorithm).

4.2.2. Preliminary memory estimation

This sections aims to assess the feasibility of Edge-Boxes integration. We consider that this is more likely to be possible in a 3D-IC architecture. For example, in a two-layer 3D-IC imaging system, the top layer would target image acquisition and pre-processing (e.g. edge extraction). Then, the bottom layer could be in charge of memory storage, ROIs detection and classification. For the moment, we approximate memory requirements for Edge-Boxes. Nevertheless, we first use "worse-case" scenario criteria for choosing the amount of segments, and bit-depths (int, float). That leads to values that are potentially high and "over-conservative". In the next sections, we show experimentally that this first theoretical value reduces significantly in real images. Nevertheless, we present our preliminary memory estimation scheme for later comparison.

In order to estimate the total memory required by Edge-Boxes, we listed all variables used in the source code (Dollar and Zitnick 2015b), and assigned an approximated[21] size formula for each of them. Moreover, we take into account that not all variables are required along the whole algorithm, thus allowing to free some space during the execution runtime. We then tried to approximate the behavior of an optimized and compiled code memory usage.

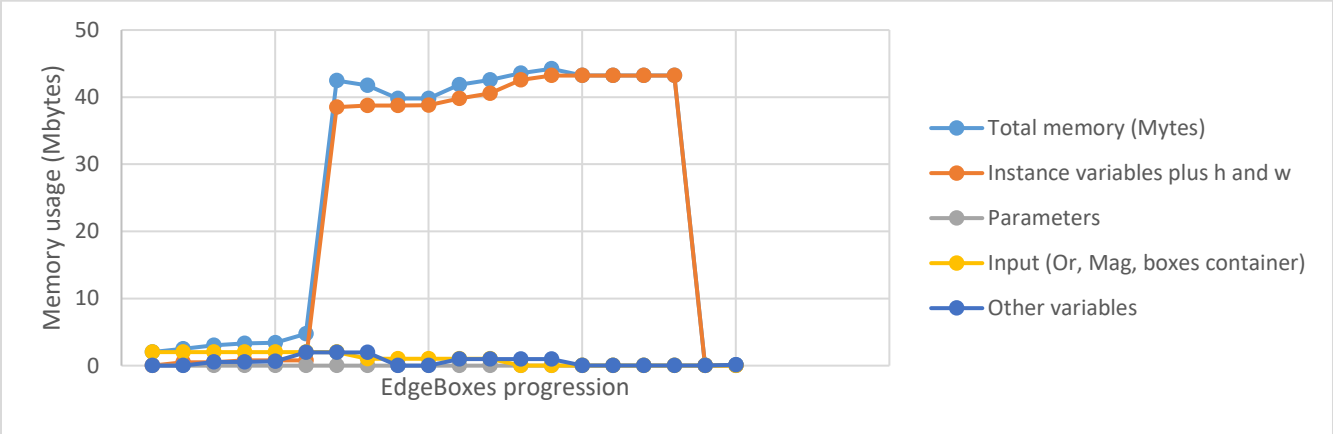


Figure 36: Edge-Boxes memory estimation while processing one image.

[20] For clarification, the term "non-maximum-suppression" is a general way of referring to algorithms that try to minimize the amount of false positives of a detection algorithm (rather it be edges, objects, or anything else) when there are close in a particular space (in our case, in the image space, closeness means spatially close). Then, if several neighboring positives for a detection algorithm have a score estimation (regardless of it is objectness, gradient magnitude, or other) the algorithm tries to only keep the local maxima and neglect the others. Notice that this is a very general definition, and its specificities depend on the context of application.

[21] In our total calculation, we take into account smaller variables and overheads related to data structures (pointers, instance variables, etc…).

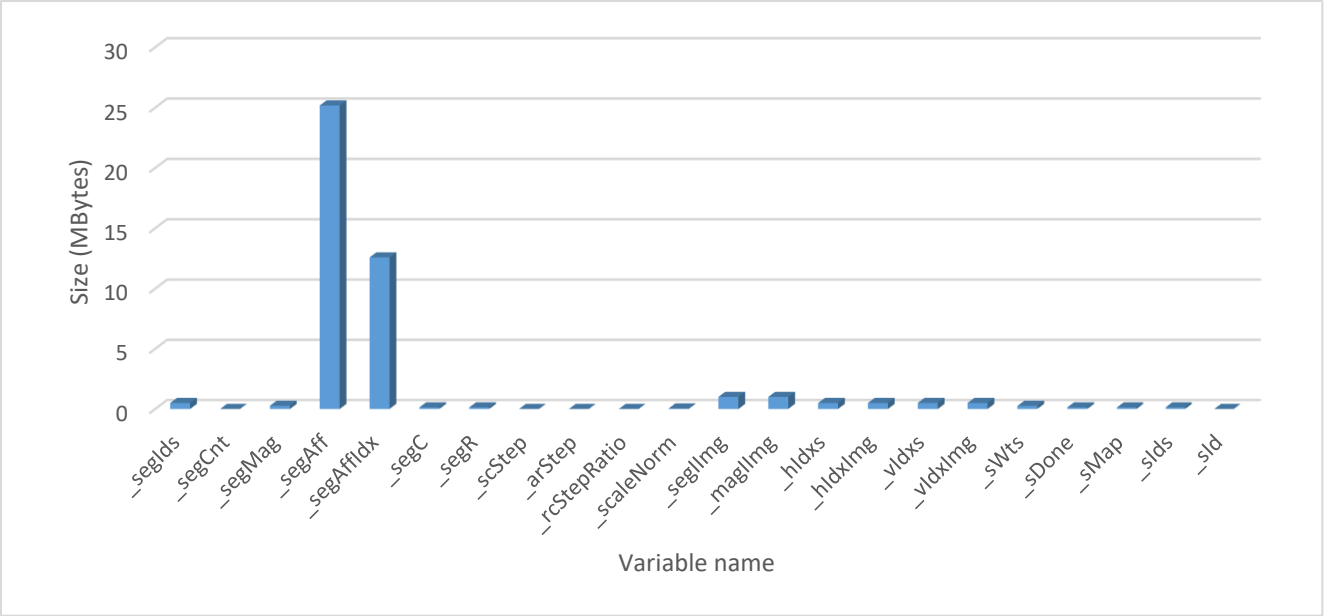


Figure 37: Edge-Boxes instance variables size estimation.

In Figure 36 shows the Edge-Boxes memory estimation while processing one image. The x axis indicates the algorithm progression divided in arbitrary checkpoints. This curve was obtained by listing all variables inside the source code and trying to estimate the optimal memory handling from the C++ compiler along the algorithm execution. From that image, we observe that there is an abrupt increase in the estimated memory. This relates to the affinities variable creation, whose size depends on the number of segments. In addition, Figure 37 shows Edge-Boxes instance variables size estimation, and derived from the Author's source code (Dollar and Zitnick 2015b). Figure 37 clarifies that, when the number of segments is big, the affinities variable takes the most important impact on memory usage. This result is reasonable since the segments affinities variable grows linearly with the amount of segments (when taking Table 4 as reference), and here we took as reference the worst case for the amount of segments. Nevertheless, we expect those theoretical results to change after we get a better statistical approximation of the practical segments amount. Then, the relative memory impact of the rest of the variables can potentially increase. We think that the memory requirements as shown in Figure 37 are too high, so going deeper for a better estimation of the segments amount is important. If the segments amount in practice is significantly less, then the affinities size will reduce as well, thus potentially making the embedded implementation feasible for embedded.

4.3. Embedding oriented gradients generation

In this section, we explain our approach to optimize the ROIs detection pipeline for embedded applications. Our strategy consists in three parts: firstly, we evaluate the viability of computing a map of oriented gradients near the matrix of pixels. That computation would exploit the pixel rolling-shutter readout with a column-parallel architecture (so that the imager calculates oriented gradients on the fly, at the rolling shutter readout speed). Secondly, we study the impact of different gradient architectures (method, encoding, and implementation) on ROI proposals performance. Finally, for each possible architecture, we extract experimentally the amount to segments generated. Then, we

derive a more realistic estimate of memory usage. In addition to the aforementioned steps, we estimate the memory usage after taking into account that, given the quantized input, reducing bit-depth inside Edge-Boxes computations could be possible as well. However, we do not test this idea experimentally, and thus we let it for further works.

As mentioned in the last paragraph, Edge-Boxes is fed with the map of oriented gradients as input. In the original paper (Zitnick and Dollár 2014), they obtained such an input with a "structured edge detection" (Dollar and Zitnick 2015a) algorithm. It used random forests, whose input was a 3-channel (color) image, and whose output was a contour map. This map is composed of two matrices: a first one containing the light intensity gradient magnitudes for every pixel (from 0 to 1), and a second one containing the corresponding local orientation (from 0 to π). The implementation available at OpenCV ("OpenCV, EdgeBoxes" 2021) used a 32 bit float type for representing them. From our perspective, this solution is too complex for a near-matrix implementation. Nevertheless, Edge-Boxes' authors mentioned that they tested the more classic Canny Edge Detection ("OpenCV, Canny Edge Detector" 2021) approach. Even though this latter approach gave a lower detection rate reported by (Zitnick and Dollár 2014), we estimated that the Canny Edge Detection was a good starting point, and potentially a good performance-complexity trade-off. In next section, we present this algorithm in more detail, as well as potential architectures that could allow to generate the contour map. Then, we compare with other approaches in the state-of-the-art literature.

4.3.1. Canny Edge Detection

We now explain the Canny Edge Detection ("OpenCV, Canny Edge Detector" 2021) algorithm in more detail, and then we start exploring how to integrate it near the matrix of pixels. The original paper (Canny 1986) includes a dense theoretical background that is not essential in this work. So we will base our explanation on OpenCV's documentation ("OpenCV, Canny Edge Detector" 2021). The algorithm takes a gray-image as input (each pixel contains one only value related to gray light intensity), and outputs an edge map. The difference with the structured edge detection, is that now the gradient magnitudes can only be either 0 or 1, and the orientations can only be 0, $\pi/2$, $3\pi/2$ and π.

In OpenCV's documentation ("OpenCV, Canny Edge Detector" 2021), the algorithm is explained as a pipeline of 4 stages: spatial noise suppression, gradient estimation, "non-maximum suppression", and "hysteresis thresholding". As a starting point, we explain briefly each on them based on ("OpenCV, Canny Edge Detector" 2021):

Spatial noise reduction

Our objective is not to provide a strict theoretical background on this subject, but rather to explain the intuition behind and its practical implementation. The point of this first stage is to reduce spatial noise caused by local intensity gradients that appear to be "erratic", and thus that may not belong to any contour. In our experiments (as will be further explained later), we observed that this spatial "noise" is potentially related to textures, and other small details in the image that are often not relevant (at least for a human) for understanding the overall morphology of different objects. Regarding the implementation, it typically involves convolving an input image with a "blur" kernel of fixed (e.g. non-trainable or handcrafted) parameters. The one suggested in OpenCV documentation

("OpenCV, Canny Edge Detector" 2021) is a Gaussian kernel[22] of size 5x5 pixels. Notice that there is not a specific Gaussian kernel for each size, since the kernel (2D-matrix) coefficients depend on a parameter σ representing the standard deviation. In OpenCV documentation, the Gaussian kernel coefficients can be obtained as ("OpenCV, Image Filtering" 2021):

$$G_i = \alpha * e^{-\frac{\left(i-\frac{ksize-1}{2}\right)^2}{2\sigma^2}}, \qquad i = 0, \dots, ksize-1$$

Equation 8 ("OpenCV, Image Filtering" 2021)

$$\sum G_i = 1$$

Equation 9 ("OpenCV, Image Filtering" 2021)

$$\sigma = 0{,}3 * \left((ksize-1) * 0{,}5 - 1\right) + 0{,}8$$

Equation 10 ("OpenCV, Image Filtering" 2021)

Where ksize is the number of kernel coefficients when taking into account one linear kernel from which the 2D Gaussian kernel can be derived (since the Gaussian Kernel is compatible with separable convolutions). For instance, for a 3x3 Gaussian Kernel:

$$\boldsymbol{k_{Gaussian}} = \begin{bmatrix} G_1 \\ G_2 \\ G_3 \end{bmatrix} [G_1 \quad G_2 \quad G_3]$$

Equation 11

Alternatively to the Gaussian Kernel, there are other types of kernel that can be applied. One possibility is to reduce the size, which may lead to a simple electronic implementation. Another possibility is to use an average[23] kernel ("OpenCV, Image Filtering" 2021): in this case, the output (de-noised) pixel value is an average of the corresponding neighbor pixels (and itself) in the input image. The amount of pixels that are "averaged" depends on the kernel size. We found this second option interesting since it could simplify the electronic implementation with respect to a Gaussian kernel. That is, because the average kernel has the same "entry" for every value, whereas the Gaussian Kernel has different values for each entry, which can make symmetric layouts more challenging. The expression of the average kernel can be found in the OpenCV documentation as ("OpenCV, Image Filtering" 2021):

$$\boldsymbol{k_{average}} = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Equation 12 ("OpenCV, Image Filtering" 2021)

Gradient estimation

This step tries to estimate the local light intensity gradient (magnitude and orientation) at every pixel from an input image $\boldsymbol{I}(x, y)$ (with x and y being pixel coordinates). The image can be colored or gray, and different algorithms accept one type or several codifications. In the original paper (Zitnick

[22]More details about the Gaussian Kernel documentation in OpenCV can be found at ("OpenCV, Image Filtering" 2021).

[23] In OpenCV's documentation, it is called the "normalized box filter" ("OpenCV, Image Filtering" 2021).

and Dollár 2014), the gradient for Edge-Boxes is estimated with the "Structured Edge Detection". It relies on a Random Forest (Dollar and Zitnick 2015a) for obtaining both gradient magnitude and direction with a float (32-bit) precision. For the Canny-edge detection, one common way is to convolve the image with two kernels $\boldsymbol{k}_{\hat{x}}$ and $\boldsymbol{k}_{\hat{y}}$, made of handcrafted coefficients, and each kernel corresponds to one image axis ($\hat{x}$, $\hat{y}$). The resulting gradients maps are $\boldsymbol{G}_{\hat{x}} = |\boldsymbol{G}_{\hat{x}}|$, and $\boldsymbol{G}_{\hat{y}} = \left|\boldsymbol{G}_{\hat{y}}\right|$ (one for each axis, and both being functions of spatial coordinates x and y). For instance, if we represent the convolution operation as $\circledast$, then $|\boldsymbol{G}_{\hat{x}}| = \boldsymbol{I} \circledast \boldsymbol{k}_{\hat{x}}$, and $\left|\boldsymbol{G}_{\hat{y}}\right| = \boldsymbol{I} \circledast \boldsymbol{k}_{\hat{y}}$. From there, the total magnitude $|\boldsymbol{G}(x,y)|$ (or simply $|\boldsymbol{G}|$) of the gradient $\boldsymbol{G}(x,y)$ (or simply $\boldsymbol{G}$) is

$$|\boldsymbol{G}| = \sqrt{|\boldsymbol{G}_{\hat{x}}|^2 + \left|\boldsymbol{G}_{\hat{y}}\right|^2}$$

Equation 13 ("OpenCV, Canny Edge Detector" 2021)

In addition, the orientation $\boldsymbol{\theta}(x,y)$, or simply $\boldsymbol{\theta}$ is:

$$\boldsymbol{\theta} = \arctan\left(\frac{|\boldsymbol{G}_{\hat{y}}|}{|\boldsymbol{G}_{\hat{x}}|}\right)$$

Equation 14 : orientation from gradient components magnitudes ("OpenCV, Canny Edge Detector" 2021)

However, notice that Canny-edge-detection "expects" $\boldsymbol{\theta}(x,y) \in [0,\pi]$, whereas $\arctan(\theta) \in]-\pi/2,\ \pi/2\,[$. Moreover, several questions rise regarding how to implement the expressions for both $\boldsymbol{G}$ and $\boldsymbol{\theta}$: firstly, which kernels to use, and how to deal with "complex expressions" for a near-to-matrix of pixels implementation, such as squaring, square root and inverse tangent. One approximation used for example by (Soell et al. 2016) is:

$$|\boldsymbol{G}| = |\boldsymbol{G}_{\hat{x}}| + \left|\boldsymbol{G}_{\hat{y}}\right|$$

Equation 15 : gradient magnitude estimation (Soell et al. 2016).

For the case of $\boldsymbol{O}$, (Choi et al. 2014) mentions the possibility of a digital "look up table" , which has as inputs both $|\boldsymbol{G}_{\hat{x}}|, \left|\boldsymbol{G}_{\hat{y}}\right|$ (represented with a particular bit-depth after analog-to-digital conversion), and outputs a discrete value representing an angle. Another idea is to perform a successive approximation in the analog domain (Choi et al. 2014): the authors explain that two successive comparisons with fixed voltage values can approximate the inverse tangent (for a low power application).

Regarding the kernels $\boldsymbol{k}_{\hat{x}}$ and $\boldsymbol{k}_{\hat{y}}$, the Sobel kernel $\boldsymbol{k}^S_{i \in \{\hat{x},\hat{y}\}}$ is for example used by (Soell et al. 2016):

$$\boldsymbol{k}^S_{\hat{x}} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \boldsymbol{k}^S_{\hat{y}} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Equation 16 : Sobel kernels ("OpenCV, Canny Edge Detector" 2021)

Another simpler kernel have been used for example by (Dalal and Triggs 2005) to estimate oriented gradients, such as:

$$\boldsymbol{k}_{\hat{x}}^{D} = |-1 \quad 0 \quad 1| \quad \boldsymbol{k}_{\hat{y}}^{D} = \begin{vmatrix} -1 \\ 0 \\ 1 \end{vmatrix}$$

Equation 17: simple linear kernel (Dalal and Triggs 2005)

Where the superscript D indicates "difference", and it is only a convenient notation for better clarity. Kernels $\boldsymbol{k}_{i \in \{\hat{x},\hat{y}\}}^{D}$ are simpler to implement, since they involve only differentiation of two pixels that are adjacent to a central one. In this work, we try to compare the performance of both cases while taking into account quantization.

(Omid-Zohoor et al. 2018) proposed to approximate $|\boldsymbol{G}_{\hat{y}}|$, $|\boldsymbol{G}_{\hat{y}}|$ as a "ratio" instead of a difference. They implemented that in the ADC, which they called the "ratio-to-digital converter" (Omid-Zohoor et al. 2018). They explained that by applying a logarithmic function on the image followed by kernels similar to $\boldsymbol{k}_{\hat{x}}^{D}$, $\boldsymbol{k}_{\hat{y}}^{D}$, then the ADC output is the logarithm of a ratio, instead of a linear difference. For instance, $|\boldsymbol{G}_{\hat{x}}| = \log(\boldsymbol{I}) \circledast \boldsymbol{k}_{\hat{x}}^{D}$, where the expression $\log(\boldsymbol{I})$ indicates that the logarithmic function spans for every single pixel-value (if considering a gray image, then $\boldsymbol{I}$ corresponds to a 2D-matrix). For instance, $\log(\boldsymbol{I})\,(x,y) = \log[\boldsymbol{I}(x,y)]$. Then,

$$|\boldsymbol{G}_{\hat{x}}(x,y)| = \log[\mathbf{I}(x+1,y)] - \log[\mathbf{I}(x-1,y)] \quad \forall\, x \in \{1,\dots,W-1\}, y \in \{1,\dots,H-1\}$$

With *W* and *H* being the image width and height (in pixels) respectively. The expression becomes:

$$|\boldsymbol{G}_{\hat{x}}(x,y)| = \log\left[\frac{\boldsymbol{I}(x+1,y)}{\boldsymbol{I}(x-1,y)}\right]$$

Equation 18 (Omid-Zohoor et al. 2018)

(Omid-Zohoor et al. 2018) explained that the obtained gradient is more robust to region light intensity changes that are not related to the presence of a feature (i.e. to changes in illumination, which can "fool" the algorithm to think that there is a feature where there is not). They illustrate it by saying that, if multiplying $\boldsymbol{I}$ along a spatial sub-region by a coefficient α, simply because such zone received more photons (illumination), then α no longer affects $|\boldsymbol{G}_{\hat{x}}(x,y)|$ due to the division. That is:

$$|\boldsymbol{G}_{\hat{x}}(x,y)| = \log\left[\frac{\alpha * \boldsymbol{I}(x+1,y)}{\alpha * \boldsymbol{I}(x-1,y)}\right] = \log\left[\frac{\boldsymbol{I}(x+1,y)}{\boldsymbol{I}(x-1,y)}\right]$$

Notice that the effect of α would remain in the case of a linear subtraction operation on the input image. Furthermore they show that such "pre-processing" in the analog domain enhances the performance of the subsequent AI processing when applying an aggressive gradients quantization. Finally they (Omid-Zohoor et al. 2018) also explain how to implement that idea by approximating the logarithmic operation with a dedicated ADC (which they call "ratio-to-digital" converter (Omid-Zohoor et al. 2018)). In this work, we take this kind or pre-processing into account, and we test it with Edge-Boxes in order to assess the global ROI performance (or objectness).

Non-maximum suppression

We explain the non-maximum-suppression (NMS) step based on OpenCV's documentation regarding the Canny Edge Detection ("OpenCV, Canny Edge Detector" 2021), and then we explain the implications of embedding it near the pixel matrix. NMS "tries" to generate "thin contours" by preserving pixels already taken as edges only if their corresponding $|\boldsymbol{G}(x,y)|$ is a local maximum along

its local direction $\theta(x, y) \in \{0, \pi/4, \pi/2, 3\pi/2\}$. However, we found several issues when trying to implement it (near the pixels-matrix, **before the** ADC). For example, one NMS iteration would be as follows:

A. Before NMS
   1. Calculate both gradient components for a 3x3 region around a pixel (x, y) and "buffer" them.
   2. Calculate gradient magnitudes from gradient components and "buffer" them.
   3. Calculate the local gradient $\theta(x, y)$ orientation at *(x, y)*, and,

B. During NMS
   4. If $\theta(x, y) = 0$, then pixel (x, y) is preserved as an edge only if $\boldsymbol{G}(x, y) > \boldsymbol{G}(x + 1, y)\ and\ \boldsymbol{G}(x - 1, y)$.
   5. If $\theta(x, y) = \pi/4$, then pixel (x, y) is preserved as an edge only if $\boldsymbol{G}(x, y) > \boldsymbol{G}(x + 1, y + 1)\ and\ \boldsymbol{G}(x - 1, y - 1)$.
   6. If $\theta(x, y) = \pi/2$, then pixel (x, y) is preserved as an edge only if $\boldsymbol{G}(x, y) > \boldsymbol{G}(x, y + 1)\ and\ \boldsymbol{G}(x, y - 1)$.
   7. If $\theta(x, y) = 3\pi/4$, then pixel (x, y) is preserved as an edge only if $\boldsymbol{G}(x, y) > \boldsymbol{G}(x - 1, y + 1)\ and\ \boldsymbol{G}(x + 1, y - 1)$.

For one single pixel, step 1 implies obtaining local gradient components of a 3x3 region around. Thus, considering any of the gradient kernels cited before, the architecture has to buffer (in the analog side) a 5x5 pixels-region in order to calculate the gradient components (if one wants to minimize pixel readings directly from the pixels-matrix). Then, step 2 implies computing the gradient components, thus performing a summation in analog domain and then buffering the resulting values. Notice that at this point, the architecture should buffer both total gradient magnitudes (for at least a 3x3 region around the pixel of interest) and local gradient components (e.g. only at pixel (x, y)). Then, the local angle is recovered from the local gradient components and buffered. After that, NMS begins in step 4. For example, a combinatorial circuit should "decide" which values of the gradient magnitude to compare with the local G(x, y), and then more electronics have to handle the comparison. Finally, the output of the local gradient magnitude output (from the ADC) is G(x, y) if the pixel is effectively an edge, or 0 for the contrary.

The paragraph above puts into evidence the potential additional complexity of implementing NMS before the ADC: several buffers are required for different computational steps, and analog or mixed-signal logic units for making decisions of which values to compare the local gradient magnitude with. Moreover, it is less likely that column-parallel parallelization could be exploited. One could argue that this could be easier to implement in the digital domain. However, the caveat is that, considering a low power ADC with aggressive quantization (for example, with magnitudes being only 0 or 1), we think that comparisons in steps 4 to 7 are likely to lose relevance and become harder to interpret. That is due to the aggressive quantization in the gradient magnitude. For instance, in the gradient in quantized to 1-bit, then the comparison of one gradient being bigger that another one becomes less meaningful, and a gradient that would normally be detected as bigger might be classified as "equal" due to quantization.

Hysteresis-thresholding

This step "tries" to estimate if pixels are (or not) edges based on their local gradient magnitude $\boldsymbol{G}(x, y)$. While edge-detection ideas for near pixels-matrix implementation, such as (Soell et al. 2016), detect edges if $|\boldsymbol{G}(x, y)| > threshold$, OpenCV's implementation (explained by them in

("OpenCV, Canny Edge Detector" 2021)) is more complex. First, all gradients are compared with two fixed thresholds T_1, T_2, with $T_2 > T_1$. Then, pixels with $\boldsymbol{G}(x,y) > T_2$ are immediately classified as edges and pixels with $T_1 < \boldsymbol{G}(x,y) < T_2$ are classified as edges only if they are connected with a pixel in which $\boldsymbol{G}(x,y) > T_2$. For all other cases, the edge-detector outputs $\boldsymbol{G}(x,y) = 0$.

From previous paragraph, we notice that this last stage presents similar issues with the previous one (non-maximum-suppression). Thus, in our approach we try to assess if a near pixel-matrix gradient architecture could work without those two lasts steps. In next section, we will continue developing the edge detection pipeline to find which one is most suitable for object detection with ROI proposals.

4.3.2. Our edge detection pipeline description

In this section, we describe in detail our pipeline for oriented gradient generation. We discuss each stage in terms of image processing, and we try to take into account IC-implementation phenomena as well. For example, we introduce more specifically where we perform quantization, and how the signal (information) goes from the pixels-matrix to the ADC.

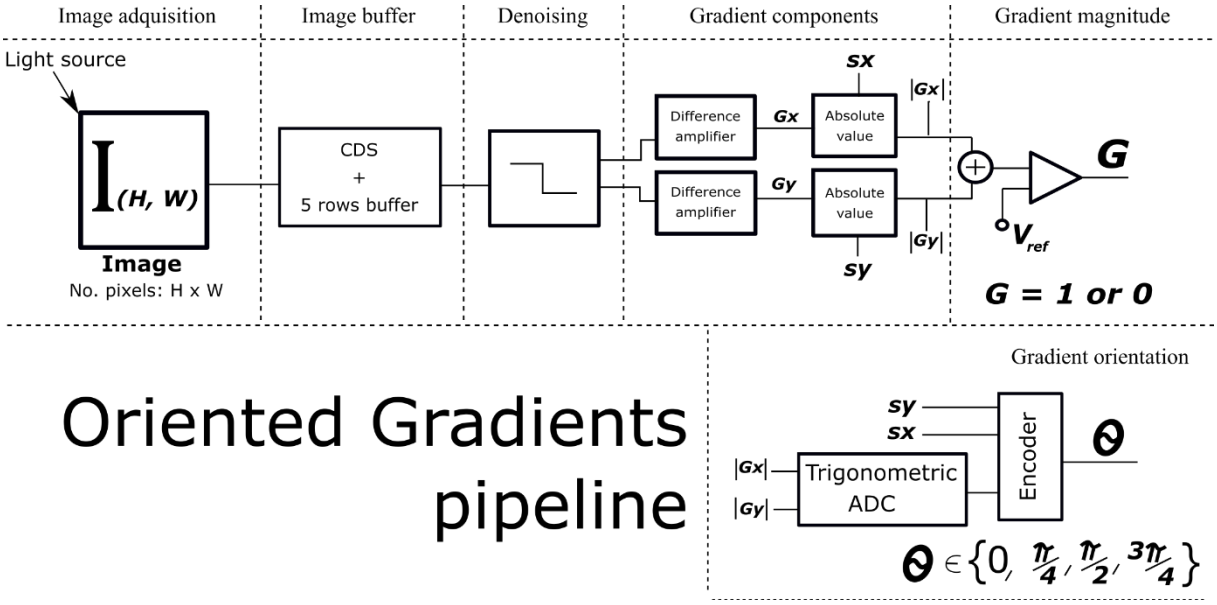


Figure 38: our proposed column parallel pre-processing pipeline for gradient computation.

In Figure 38 we represent a series of stages for near pixel-matrix edge-extraction (i.e. extraction of oriented gradients at individual pixels and which are higher than a reference threshold). It is important that this is not the first architecture proposed to perform this task, thus we will make emphasis on differences with the state to art when required. Continuing with the overall description, our approach targets column-parallel computations, instead of inside-pixel architectures as in (Bose et al. 2019). Our motivation is to be able to separate pixel-matrix design from pre-processing, then allowing optimizing both independently. Moreover, in-pixel architectures have important issues with fill-factor, fixed-pattern-noise and pixel complexity: all of which make it more difficult to implement (and model) a pre-processing pipeline like the one we chose.

According to Figure 38, the different stages are:

1. Image-acquisition (a standard *H x W* pixels-matrix with column parallel read-out).
2. Correlated double sampling, and 5x5-pixels circular-image-buffer
3. Average 3x3-kernel de-noising.
4. Gradient components calculation with kernels as in Equation 17.
5. Gradient and orientation:
    a. Gradient magnitude estimation as in (Soell et al. 2016).
    b. Orientation estimation with the approach suggested by (Choi et al. 2014)[24] (with successive approximations).

The remainder of this section describes the behavior of each of the stages, starting step 2 in the list above.

CDS and circular buffer

We think that correlated-double-sampling (CDS) is important to mitigate FPN and temporal noise, especially because the architecture is multi-staged, and noise "propagates" during pre-processing. We took the CDS plus buffering architecture from (Young et al. 2019), and we slightly modified it to adapt for including the de-noising (include capacitances to change kernel size). As they (Young et al. 2019) explain, with the right amount of capacitances, this block also allows performing "binning" if needed. The importance of a circular buffer is that gradient calculations depend on values of neighboring pixels, and we wanted to avoid at maximum to read pixels directly from the matrix several times. Indeed, different values of pixels are required several times, and one gradient value (magnitude plus direction) depends on 25 values in the original image. For example, in Figure 39, the yellow (thick border) rectangle in (a) englobes original values needed for computing the de-noised value $I(4,4)$ in (b), which is needed for gradient computation at pixel (4, 5). The thick borders in (b) represent all de-noised values needed for computing the Gradient G at pixel (4, 5) (the gradient matrix is not shown). At the same time, the light-blue background in (a) represent all original values used for computing one single gradient value at pixel (4, 5). That is, one entry in the gradient matrix "expands" to a 5x5 matrix in the original pixel-values-matrix. Now, we center our attention at location (4, 5), where our goal is calculating the gradient. From that figure, we observe that due to the 2-stages (image acquisition and de-noising), this gradient value depends on values from a 5x5-matrix in the original image. That is, we want to buffer those 25 values for a simple gradient computation. Moreover, when computing gradient at other locations, like at (3, 5), original-pixels-values that were used before are required again. That is, the buffer has to "keep" pixels values while they are still needed.

[24] We represented it as a trigonometric ADC for clarification. However, (Choi et al. 2014) did not implemented this idea for an ADC. In fact, they use values already in the digital domain, transferred it to the analog domain, and then went backwards to the digital side. All of that only for calculating the gradient orientation.

Figure 39: Illustration of why a simple gradient computation depends upon 25 original-image-values.

De-noising

In our simulations results (depicted in detail in the benchmarks chapter), we found that de-noising has an important impact in the amount of pixels that are detected as edges. Moreover, this stage improves the performance for ROI proposals with Edge-Boxes and with quantized gradients. We think this relates to the mitigation of high spatial gradients corresponding to textures and not to overall topologies. However, when de-noise is not present, the elevated amount of edges increases the amount of clusters of edges in Edge-Boxes, which increases memory and runtime as, will be shown in chapter 6. Thus, de-noising allows to decrease memory and runtime, at the same time that ROIs are obtained with the same or even better accuracy.

We selected an average kernel because it was the simplest to implement, and our decision is supported by results that will be presented at the benchmarks chapter (chapter 6). Indeed, in our tests, a more complex kernel type did not change significantly the accuracy for ROI proposals.

Another important aspect is that we have not found another work which proposes a column-parallel **staged-pipeline** pre-processing including both de-noising and gradient computation (i.e. without a "standard computational unit" or a standard addressable memory). This column architecture exploits parallelism and computes gradients on the fly. In chapter 5 we go deeper in how we propose to achieve that. We found works, for example (Hsu et al. 2019; Soell et al. 2016; Young et al. 2019), proposing analog computing architectures for image pre-processing. However, they only computed one single stage (either de-noising or pixels differences) before the ADC, or they implemented a more complex computational unit resembling a standard processor. In that case, it is less obvious to exploit parallelism and compute gradients on the fly.

Gradient components

We selected the kernels showed in Equation 17 because of their simplicity, and since results (in the benchmarks section) support that this kernel is "good enough" for ROI proposals. Indeed, a more complex type of gradient, such as a Sobel (Equation 16), did not lead to significant improvements. Hence, we propose calculating each gradient component with two separate blocks (one for each component) as proposed by (Soell et al. 2016). In their work, however, they did not use a de-noising stage previous to the gradient extraction. We also keep the idea from (Soell et al. 2016) of calculating the absolute value for each component for the next stage, yet we add two additional signals respect to (Soell et al. 2016), ***sx*** and ***sy***, the gradient signs, that are important for the angle.

Gradient magnitude

Similarly to (Soell et al. 2016), we simply "add" the gradient magnitude from each component and we compare the sum with a fixed threshold ($\boldsymbol{V_{ref}}$). This comparison outputs either 1 (high) or 0 (low) values, thus quantizing the magnitude to 1 bit values. During our experiments, we often observed that many pixels (between 80 to 90%) are not detected as edges (when the threshold is set correctly). Then, we concluded that the architecture should compute the angle only when the magnitude equals 1 (high). That could save power and speed up gradients computation. For instance, by exploiting the edge map sparsity: if the gradient magnitude is 0, the angular calculation is not performed, and only 1 bit is read from the ADC. Two possible ways of exploding that (that are let to be studied in further works) are to reduce the trigonometric ADC activity by approximately 90 % for angular calculation. Secondly, to allow the system to run in a simpler (probably faster) mode by only detecting edges and not angles. That could be attractive for autonomous systems with adaptive behaviors.

Angle computation

For the angular computation, we propose to adapt the block from (Choi et al. 2014) to carry out the inverse tangent approximation. The principle consists in comparing the ratio Gx/Gy to tan(θ_T) where θ_T is a threshold angle that can be adjusted. As an example, if we want to discriminate 4 angles from 0 to 180°, you need the comparison results from:

- Comp1 : |Gy| > |Gx|.tan(22.5°), meaning θ >22.5° if Gy/Gx> 0, or θ <177.5° if Gy/Gx< 0,
- Comp2 : |Gy |> |Gx|.tan(67.5°), meaning θ >67.7° if Gy/Gx> 0, or θ <132.5° if Gy/Gx< 0,

For simplicity, we call this block "trigonometric ADC plus an encoder" in Figure 38. From (Choi et al. 2014) , we took into account several differences that we had to adapt for our case. Firstly, in their implementation (Choi et al. 2014) the angular computation was after the ADC (thus, in the digital side). Then, they had to include the digital to analog conversion. Our implementation will be simpler as we already are in the analog domain. Secondly, since the angle depends on gradient-components signs, they used the previously obtained signs from the gradients calculation in the digital side. In our case, we use the sign signals output from the analog absolute value function instead, based on the architecture proposed by (Soell et al. 2016). Finally, for our implementation, the whole circuitry can be simplified, since they (Choi et al. 2014) projected the angle into 9 bins, whereas we are interested in 4 angular values (or 4 bins).

4.3.3. Dynamic range improvement

The ratio-to-digital converter architecture proposed by (Young et al. 2019; Omid-Zohoor et al. 2018) is more robust to high-dynamic-range effects related to gradient-components-quantization. They (Young et al. 2019; Omid-Zohoor et al. 2018) approximated logarithmic gradient-component-conversion with two successive comparisons. To explain this process, we take into account one gradient component along x (it could be y as well) $\boldsymbol{G_x}(x, y)$. It depends on image values (not de-noised, in the case of (Young et al. 2019; Omid-Zohoor et al. 2018)) $\boldsymbol{I_x}(x-1, y)$ and $\boldsymbol{I_x}(x+1, y)$. In the analog domain, those two image values are voltages corresponding to (respectively) are $\boldsymbol{V_{i-1}}$ and $\boldsymbol{V_{i+1}}$. Then, the first comparison is $\boldsymbol{V_{i-1}} < \boldsymbol{V_{i+1}}$. If true, then the second comparison is $\boldsymbol{V_{i-1}} < 1/2\, \boldsymbol{V_{i+1}}$. If the first comparison result is false, the second one is similar, but the sub-indexes are switched. Then, for their case (Young et al. 2019), gradient components where obtained with Table 6:

Table 6: truth table suggested by (Young et al. 2019) for approximating logarithmic 2-bit gradient-components.

Comparison 1	Output 1	Comparison 2	Output 2	$\boldsymbol{G_{i=x,y}}$
$\boldsymbol{V_{i-1}} < \boldsymbol{V_{i+1}}$	1	$\boldsymbol{V_{i-1}} < 1/2\, \boldsymbol{V_{i+1}}$	1	1
			0	0
	0	$\boldsymbol{V_{i+1}} < 1/2\, \boldsymbol{V_{i-1}}$	1	-1
			0	0

From Table 6 presents truth table suggested by (Young et al. 2019) for approximating logarithmic 2-bit gradient-components with successive approximation (comparison). From Table 6, we observed that this dynamic range improvement is related with comparing voltages (light intensities) that are spatially close. The idea of a fixed threshold for "simple or linear gradients" is changed for a "fixed factor", which is multiplying one on the two voltages. We then wondered if we could apply a similar idea to our pipeline (depicted in Figure 38). In order to do so, we also observed that:

$$\boldsymbol{V_{i-1}} < \frac{1}{2}\boldsymbol{V_{i+1}} \Rightarrow 2\boldsymbol{V_{i-1}} < \boldsymbol{V_{i+1}} \Rightarrow \boldsymbol{V_{i-1}} + \boldsymbol{V_{i-1}} < \boldsymbol{V_{i+1}} \Rightarrow$$

$$\boldsymbol{V_{i+1}} - \boldsymbol{V_{i-1}} > \boldsymbol{V_{i-1}}$$

On the left hand side of this expression is the linear gradient component. What has changed is the right hand side, which is a variable voltage (representing light intensity) instead of a fixed value. From here, we consider a second approximation, by changing the threshold from $\boldsymbol{V_{i-i}}$ to $\boldsymbol{V_i}$:

$$\boldsymbol{V_{i+1}} - \boldsymbol{V_{i-1}} > \boldsymbol{V_i}$$

$$\boldsymbol{G_x} = \boldsymbol{V_{x+1}} - \boldsymbol{V_{x-1}} \quad , \quad \boldsymbol{G_y} = \boldsymbol{V_{y+1}} - \boldsymbol{V_{y-1}}$$

Then, if we apply the condition the gradient magnitude, we obtain:

$$|\boldsymbol{G_x}| + |\boldsymbol{G_y}| \approx |\boldsymbol{G}| > \boldsymbol{2V_i}$$

We interpret that last condition as: in order to detect an edge at pixel location *(x, y)*, the estimated local gradient magnitude has to be higher than twice the local light intensity value. We did not find any particular justification for the factor of 2. Then, the last inequality can be generalized as:

$$|\boldsymbol{G_x}| + |\boldsymbol{G_y}| > \alpha \boldsymbol{V_i} \Rightarrow Edge\ detected$$

Equation 19

We call the gradients detected by last inequality as "*relative gradients*", since its magnitude depends on the local light intensity value instead of a fixed threshold. Again, only if an edge is detected, then we proceed to compute the angle. Another important difference, which will be relevant in the benchmarks section, is that estimating the angle based on "logarithmic-gradient-components" is not the same as estimating it with "linear gradient components". The reason is that, theoretically, estimating the angle from linear gradient component is an approximation of the inverse tangent of the

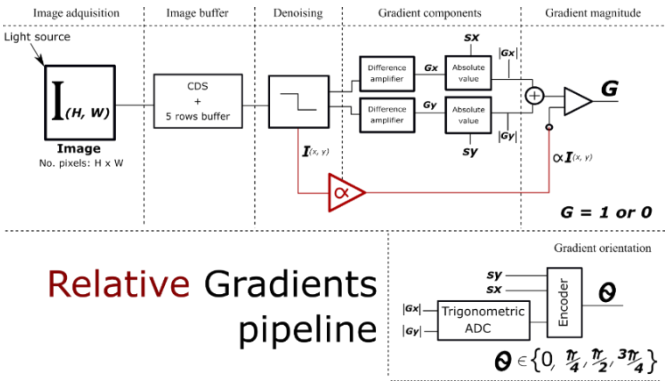


Figure 40 : adaptation from Figure 38 for relative gradients. The architecture segments added are colored in red.

ratio of linear gradients. Nevertheless, obtaining the angle from logarithmic components, is an approximation of the inverse tangent of the logarithm of the ratio of the components. Those two functions are not (to our knowledge) to be mathematically equivalent. In addition, based on results that will be shown and explained in the benchmarks section, using linear gradients for angle approximation gave better results in overall for ROI proposals with Edge-Boxes. Notice that "logarithmic-gradient-components" refers to the architecture proposed by (Young et al. 2019), in which they only calculated angles based on two gradient components in the X and Y axis. Nevertheless, if four gradient components were calculated (X and Y axis and the 2 diagonals), the angle assessment could be reduced to only finding the maximum of the four gradients (which was not studied in this work, since we found that it was interesting after discussing the results).

4.4 Conclusions of chapter 4

In this chapter we discussed the feasibility of ROI proposals generation in an embedded (hardware-constrained) architecture. We observed that, due to complexity of state of the art algorithms, it is not obvious to implement them near the sensor. However, a good candidate is the algorithm called Edge-Boxes. We decorticated this algorithm to have a first assess of memory usage as first estimation of implementation viability. We found that using worst case criteria gives rather elevated memory requirements (~50 Mbytes), but that may not be realistic, since in practice a much smaller number of contour segments could be required (for now we let this as an open question until we address it our benchmarks chapter). After that, we explained different methods for obtaining, at the column level, the Edge-Boxes input: the edge map. Moreover, we selected a specific pipeline for a possible implementation based on power constraints, but also for optimizing the A.I. performance. In the next chapter, we will dig into each block of our proposed gradient computation scheme by presenting possible electronics schematics, functioning, and power consumption approximations. In the next chapter, we will start by decoupling all the processing in stages.

Chapter 5: Embedded Edge Extraction Circuitry

In this chapter, we focus on describing the different stages for edges extraction, which we call Origrad, at the schematic level. Our principal objective is to get first approximations for power consumption, bandwidth and noise. Those approximations are based on the behavioral functioning of each stage, associated with estimated parameters (stray capacitances, intrinsic transistor gain, …). As presented in chapter 4, and for the sake of clarity, we divide the whole pipeline into five stages. The first one, the correlated doubled sampling and circular buffer circuit, is the one proposed by (Young et al. 2019)*. The only difference is that we added two sampling capacitors to buffer two more pixels per column (and again 1 more if binning is required). We proposed the next two stages, the low pass filter and the intermediary buffer, inspired from typical sampling and capacitive charge recombination circuits. We took the circuitry from* (Soell et al. 2016) *for the final two stages: difference plus absolute value, and summation plus quantization. The only difference we made was to adopt sampling-capacitances and switched-capacitor topologies.*

For our analysis, we assume that the matrix of pixels outputs a gray intensity image of size H x W (rows times columns). In addition, we foresee that there is one pipeline circuit for each column. Nevertheless, if area constraints impose it, the circuitry could be further adapted so that several image columns share one pipeline circuit.

5.1. Circuit stages

5.1.1. Correlated double sampling and first buffer

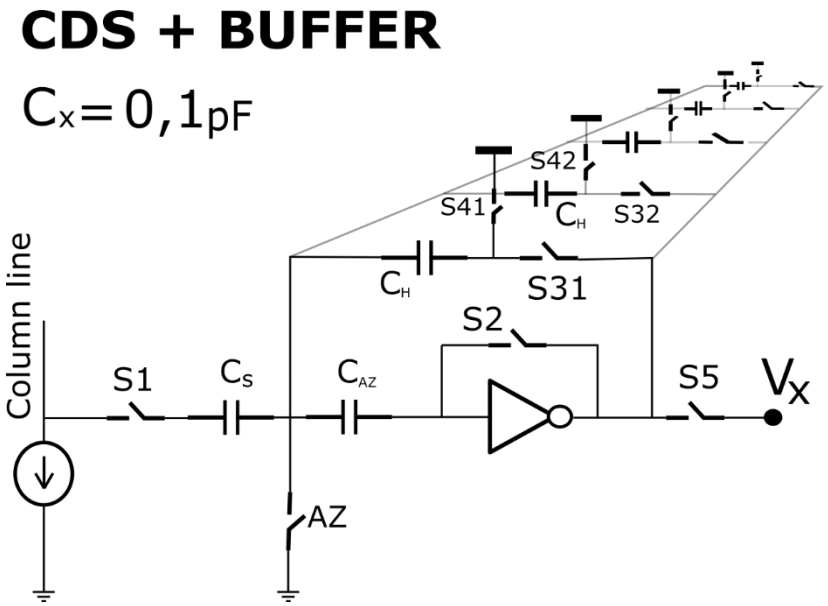


Figure 41 : Correlated double sampling and buffer circuitry proposed by (Young et al. 2019).

This circuit was taken from (Young et al. 2019), related to the pipeline in Figure 5, with the only difference being that we use 5 capacitances C_H instead of 4. If n x n binning is desired, a sixth capacitance C_H should be added in parallel to the circuit in Figure 41. The reader can find more details about n x n binning with this architecture in (Young et al. 2019). In addition, as explained by (Young et al. 2019), this circuit performs a CDS, removing the pixel reset noise and the low-frequency noise, by sampling onto any of the capacitances C_H the difference between the pixel-signal value and the pixel-reset value. Switches S31 to S35 determine the capacitance at which each pixel value goes. Each C_H samples one row at a time and a digital-state-machine controls the row corresponding to each capacitance such that the circuit behaves as a circular buffer of 5 rows. 5 rows are needed as explained in chapter 4. Taken into account that this happens for every column, the whole first preprocessing stage buffers 5 complete rows at the same time during the preprocessing. Then, when a new row is read and sampled, the rest of the pipeline computes new features, a row output (of features) is generated, and then a new row is sampled again (overwriting a row that is no longer used for preprocessing). In the next paragraph, we try to estimate the power consumption for this stage.

Let fps be the number of frames per second output by the image sensor, and f_{s1} be the pixel-value sampling-frequency at C_s during reading. Since there is one pipeline circuitry per column, then $f_{s1} = H \cdot fps$, where H is the number of rows in the original matrix of pixels. We approximate the power for this stage, per column, as $P_1 = \alpha \cdot V \cdot I$ (idle factor, times voltage, times current), so P_1 is in units of $W/column$.

$$P_1 = \alpha I_{inv} V_{inv}$$

Equation 20

In last equation, α represents a sort of "activity factor" (similar to digital electronics) which takes into account that the inverter biasing can be "set to off" when it is not in use (duty-cycling). This is not rare since, for example, autonomous systems can lower the frame rate in order to save energy. Moreover, for a whole object detection pipeline the feature map acquisition could use just a small portion of the whole A.I. processing time.

5.1.2. Low Pass Filter

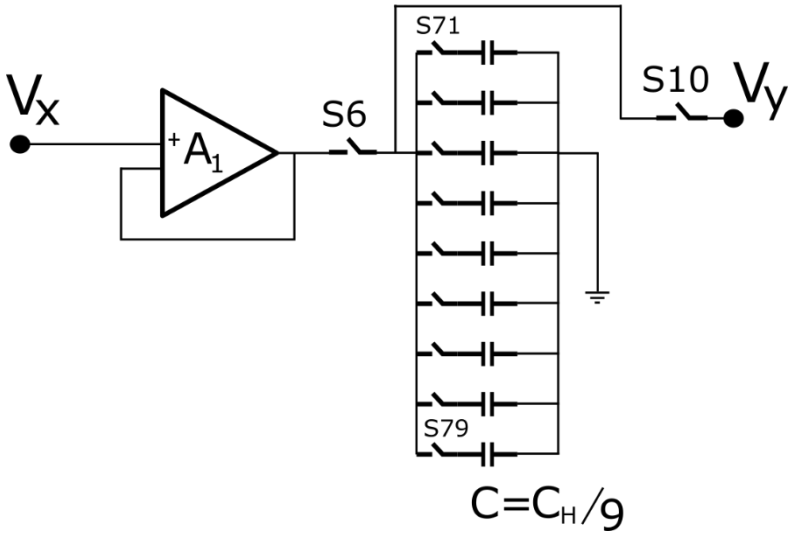


Figure 42 : image spatial-low-pass (blur) filter with an averaging kernel.

The aim of this circuit is to convolve (on the fly) the gray-intensity-image with a 3x3 averaging kernel. This stage adds a complexity and power overhead respect to other edge-detectors such as (Young et al. 2019; Soell et al. 2016). Nevertheless, in the benchmarks (chapter 6) we will discuss further about the interest of this stage. One can note that this stage is not critical as the edge detector will still "work" even if the average kernel is not applied. Indeed, the objective of this stage two is to enhance or optimize the A.I. performance vs. runtime trade-off.

The principle of functioning is the following: during the sampling phase, switch S6 is closed whereas S10 is open. Switches S71 to S79 select which capacitance the value is sampled on, and the mapping between the sampling capacitance and the row, column positions in the original image is carried by a digital-state-machine. Thus, a 9 x 9 window of the original image is sampled in those capacitances. For the second stage, S6 opens and S10 closes, such that the output voltage at node y represents an average of sampled values. In next paragraph we start estimating the power consumption.

Let f_{s2} be the frequency at which values are sampled in any of the switched capacitances in *Figure* 42. Then, this frequency has to be nine times the frequency at which values are sampled in next and previous stages (e.g. nine sampled "pixel positions" in this stage correspond to "just one" averaged value). Then $f_{s2} = 9 \cdot f_{s1}$, and:

$$P_2 = \alpha I_{amp} V_{amp}$$

Equation 21

Where I_{amp} and V_{amp} correspond, respectively, to the current and voltage biasing for the unity gain buffer A_1.

5.1.3. Second buffer

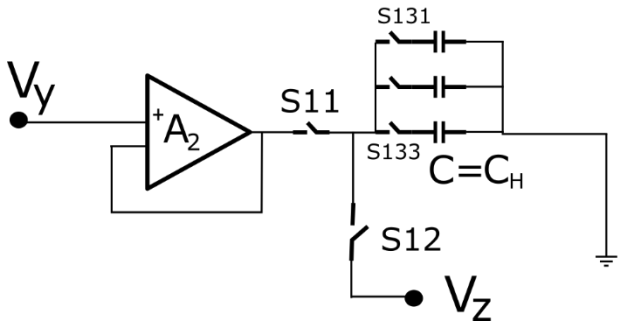


Figure 43: intermediary buffer schematic.

This stage (Figure 43) allows to store 3 averaged values corresponding to 3 pixels from 3 consecutive rows (when thinking of the original image). This obeys the pipeline introduce in chapter 4, with this intermediary buffer between the de-noising and the subtraction. Since it is implemented in every column, three whole rows are sampled in this stage as the next stage carries out the gradients components computation with a 3x3 kernel. The intermediary buffer performs a first sampling, switch S12 is open while S11 is closed. Switches S131 to S133 select the corresponding capacitance for sampling the denoised pixel value. This is controlled by a digital state machine on the periphery, since switching signals are the same for all columns. For the reading phase, switch S11 is open, while switch S12 is closed. Again, switches S13i select the value to read.

By keeping a logic similar to last stage, one value sampled at any of the capacitors in Figure 43 corresponds only to one pixel address at the pixel matrix. Then, the value storing frequency is $f_{s3} = f_{s1}$, with f_{s1} the frequency of sampling at C_s in Figure 41. Then, the power consumption is:

$$\mathrm{P}_3 = \alpha I_{amp} V_{amp}$$

Equation 22

Once values are « de-noised », they are ready to be used for gradient components computation. Then, next stage « reads » 2 values sequentially to compute the difference and the absolute values of the gradient components. In addition, notice that in last chapter, stages 2 (low pass filter) and 3 (intermediary buffer) are merged into one single block. Indeed stage 3 comes specifically from implementation details only. We decided to separate stages 2 and 3 only to facilitate our analysis.

5.1.4. Difference and absolute value

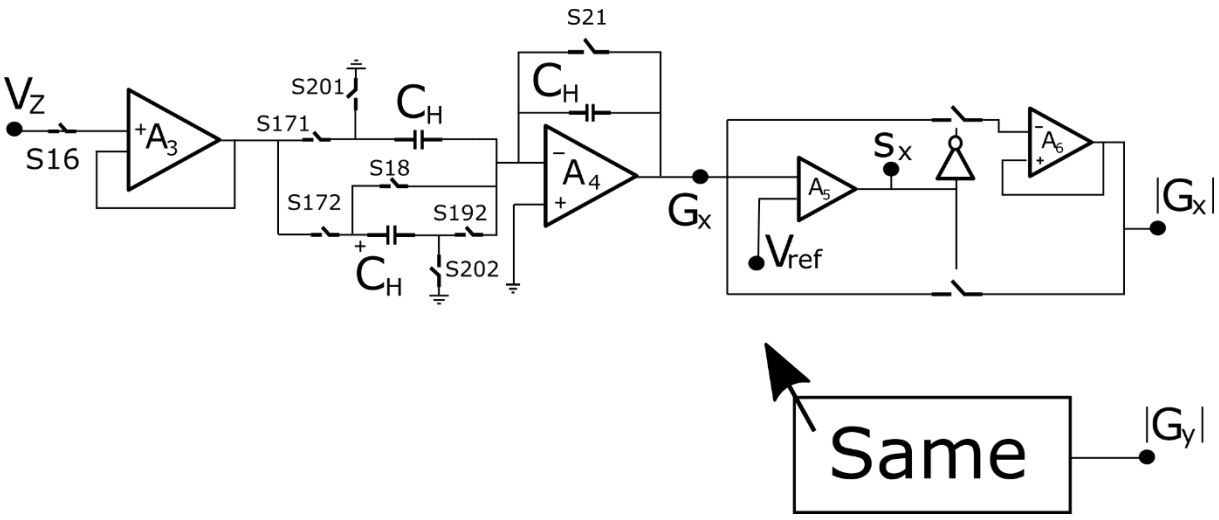


Figure 44 : schematic for computed gradient magnitude components.

This stage can be accomplished with a sequential computation of the two main axes components. Nevertheless, if area constraints allow it, the same architecture can be repeated twice. We take the second option for our further analysis, since it is interesting to explore a more aggressive parallelization. This stage is based on the publication from (Soell et al. 2016), where the subtraction was changed for taking switched-capacitance amplifiers into account. The absolute value circuit was left completely as presented in (Soell et al. 2016).

The functionality of the circuit in Figure 44 is as follows: firstly, at sampling stage (Figure 45), switches S16, S21 are closed, while S201, S202 and S18 are opened. Switches S171 and S172 select the capacitance for sampling the previously de-noised value hold in the intermediary buffer. Two values are required to compute each component. Then, corresponding switches to read/sample each value are controlled by an external digital-state-machine placed on the periphery. Once the two values are sampled, the second stage is the subtraction and absolute value computation. There (Figure 46), switches S16, S171, S172, S192 and S21 are opened, while S201, S02, S18 and S191 are closed. Notice that S18 allows changing the polarity of the read value from the capacitor below it in *Figure* 44. Then, during subtraction, charges from both capacitors at the left of amplifier A4 (*Figure* 44) are subtracted and the resulting charge is stored in the capacitor below S21. Then, one gradient component (as an analog voltage) has been computed, and the rest of the circuitry carries the absolute value function as explained by (Soell et al. 2016).

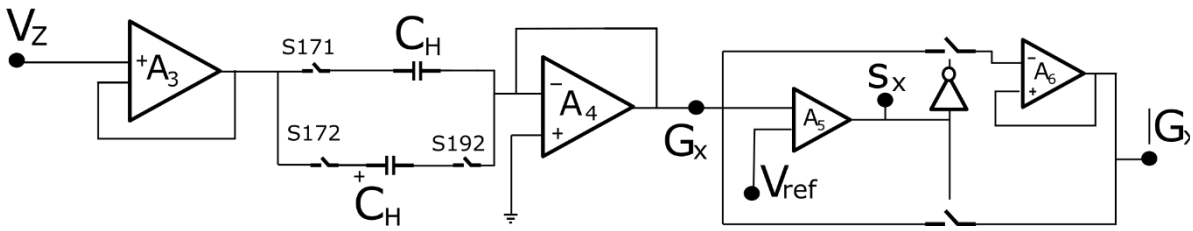


Figure 45 : equivalent difference and absolute value circuit from Figure 44 in sampling phase.

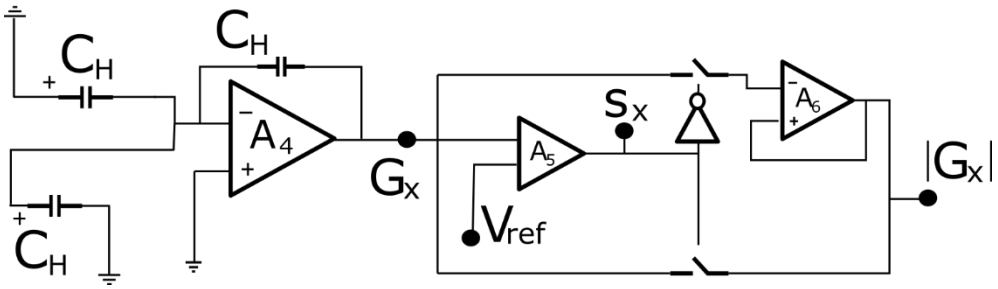


Figure 46 : equivalent difference and absolute value circuit from Figure 44 in amplification stage.

For estimating the power consumption, we assume that all amplifiers are biased with the same parameters. Moreover, for each pixel address in the original image, two values are sampled for the gradient calculation at capacitors to the left from A4, whereas the one below S21 charges only once per subtraction. Then $f_{s41} = 2f_{s1}$ and $f_{s42} = f_{s1}$. Moreover:

$$P_4 = 2 \cdot \left(\alpha 5 I_{amp} V_{amp}\right) = 10\alpha I_{amp} V_{amp}$$

Equation 23

Once the two components are calculated in parallel, they can be used to calculate the total gradient magnitude in next stage.

5.1.5. Summation and quantization

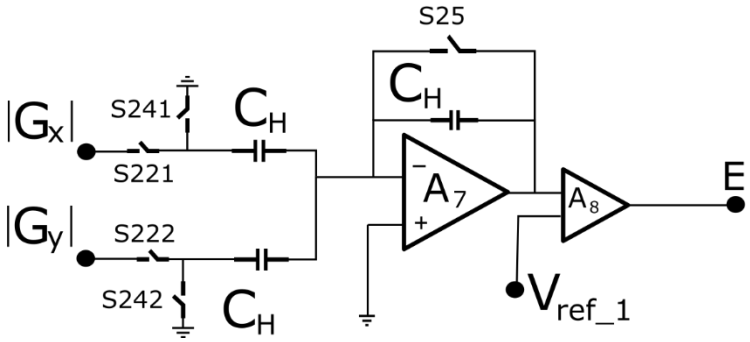


Figure 47: circuit schematic for the summation and quantization stage.

This circuit computes (see Figure 47), firstly, the summation of both gradient-components absolute-values. Secondly, it estimates the 1-bit gradient magnitude by comparing the later summation with a fixed reference (Vref_1 in *Figure* 47). The functioning of this circuit is as follows: for the sampling phase, switches S241 and S242 are opened, while the rest are closed. Then, the abs-component-values are sampled at capacitances at the left of amplifier A7 and the capacitor above it is (ideally) set to zero voltage (charge). For the next stage, switches S221, S222 and S25 open, while switches S231, S232, S241, S242 are closed. Then, the charge from capacitors at the left of A7 is « transferred » to the capacitance above it, completing the amplification (summation). The summation is then compared by means of a simple comparator amplifier. Then, $f_{s5} = f_{s1}$.

The power consumption can be thus estimated as:

$$P_5 = 2\alpha I_{amp} V_{amp}$$

Equation 24

5.1.6. Angular computation

Figure 48 shows the circuit schematic, inspired from (Choi et al. 2014), for sorting the gradient angle among the four values {0°, 45°, 90° and 135°}. The principle of functioning is as follows: during sampling time, switches 611, 612, 621, 622 are ON for sampling the gradient component values. Also, S641, S642 are ON for reset of capacitors C1 and C2. |Gx| is sampled twice, at C01 and C02, and the analog happens for |Gy|. During the logic phase, all switches previously ON pass to OFF, and switches S631, S632 pass from OFF to ON. The result is that two key comparisons are performed, in order to assess the angle. Compared to (Choi et al. 2014), since we only needed 4 angular values (2 bits), the schematic and functioning was simplified, while preserving the same key ideas. During the logic phase, the two logic comparisons performed are $|G_y| > |G_x| \cdot \tan(67.5°)$, and $|G_x| < |G_y| \cdot \frac{1}{\tan(22.5)}$. Similarly as suggested by (Choi et al. 2014), those values are determined by C01, C1, C12 and C2. The result from those two comparisons are fed to an encoder, along with logic signals sx, sy, which contain the gradient components sign. Figure 49 shows the truth table for projecting the angle on only the first quadrant, according to the results of the two comparisons mentioned before. For including the second quadrant as well, an extended truth table (considering sx, sy) is showed as well.

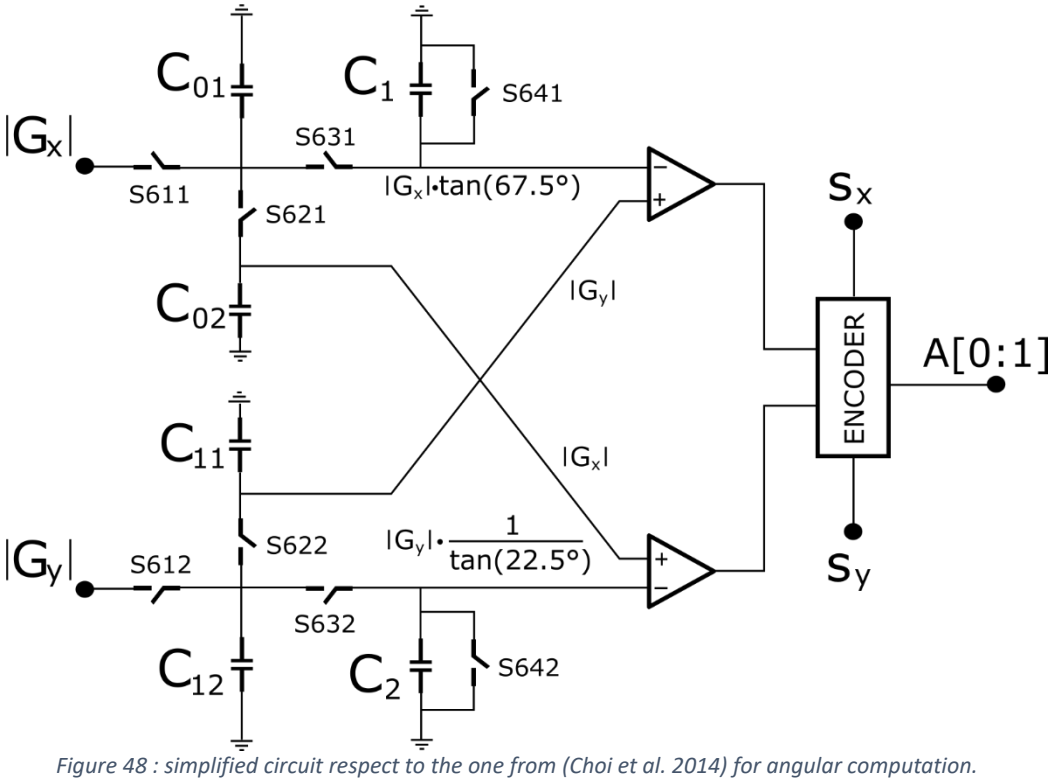


Figure 48 : simplified circuit respect to the one from (Choi et al. 2014) for angular computation.

Regarding the power consumption, we approximate it by taking into account the power from the two amplifiers (comparators). We neglect the power consumed by the digital part (the encoder). The bandwidth of both comparators is $f_{s6} = f_{s1}$.

$$P_6 = \alpha(2 \cdot V_6 I_6)$$

Equation 25

$$\frac{|G_y|}{|G_x|} > \tan(67.5°) \Rightarrow |G_y| > |G_x| \cdot \tan(67.5°) = B1$$

$$\frac{|G_y|}{|G_x|} < \tan(22.5°) \Rightarrow |G_x| > |G_y| \cdot \frac{1}{\tan(22.5°)} = B2$$

Quadrant I truth table

B1	B2	ϴ
0	0	45°
0	1	0°
1	0	90°
1	1	X

Quadrants I, II truth table

B1	B2	s_x	s_y	ϴ	A0	A1
0	0	0	0	45°	0	1
0	1	0	0	0°	0	0
1	0	0	0	90°	1	0
1	1	0	0	X	X	X
0	0	0	1	135°	1	1
0	1	0	1	0°	0	0
1	0	0	1	90°	1	0
1	1	0	1	X	X	X
0	0	1	0	135°	1	1
0	1	1	0	0°	0	1
1	0	1	0	90°	1	0
1	1	1	0	X	X	X
0	0	1	1	45°	0	1
0	1	1	1	0°	0	1
1	0	1	1	90°	1	0
1	1	1	1	X	X	X

Figure 49 : logical expressions and related truth tables for angular computation with the circuit from Figure 48.

5.2. Condensing power into a single formula

The total power is the sum of all stages multiplied by the number of columns. We are also interested at the "energy per pixel per frame", which is a figure of merit reported by other works such as (Young et al. 2019). Then,

$$\frac{P_{total}}{W}\left[\frac{Watts}{Column}\right] = P_1 + P_2 + P_3 + P_4 + P_5 + P_6$$

Equation 26

$$\frac{P_{total}}{W} = (\alpha I_1 V_1) + (\alpha I_2 V_2) + (\alpha I_3 V_3) + (10\alpha I_4 V_4) + (2\alpha I_5 V_5) + (2\alpha I_6 V_6) = \alpha(I_1 V_1 + I_2 V_2 + I_3 V_3 + 10 I_4 V_4 + 2 I_6 V_6)$$

Equation 27

In last equation, we assume that the inverter amplifier in first stage is also biased with the same parameters as the rest of amplifier in other stages. For selecting the factor α, we first set it to 1 and observe the implications of doing so. Some calculations are presented below:

Table 7: power estimation when taking $V_i = 1{,}5\ V, I_1 = 1\ \mu A,\ H = W = 500\ pixels$, and $fps = 60\ frames/s$.

alpha	Total power (mW)	E per pixel (pJ/pix)
1	12,750	850,0
0,1	1,275	85,0
0,01	0,128	8,5
0,001	0,013	0,9
0,0001	0,001	0,1

In Table 7, we illustrate the effect of alpha in reducing the energy per pixel, assuming alpha does not depend on the bandwidth. We chose arbitrary values for the bias, based on values reported by (Young et al. 2019) for the CDS plus circular buffer stage. *Notice that those are not the final theoretical values, since they have to be consistent with the required bandwidth*. That will be derived in next section. Also, the energy per pixel is calculated by dividing the total power by the frame rate and the total number of pixels W times H. We chose a typical value for the fps, but again, this value is just illustrative, and it will be derived in next section. From one can observe that for a α equal to 1, the energy per pixel is one order of magnitude higher than the one reported by (Young et al. 2019). Then, we observe that power can be reduced either by duty-cycling, or by reducing the biasing (voltage or current). Indeed, that will be our objective in next subsection.

5.3. Estimation of the bias current

In last section, the bias current was selected as a rounded value from the CDS plus buffer stage proposed by (Young et al. 2019). This selection, however, is still arbitrary and here we try to go deeper in how to assess better the expected order of magnitude for the biasing current. Please consider that our objective is to study a rather simplistic test scenario for delimiting the reasonable range of the biasing current I_d. Up to now, we let this current for all previously described stages equal to one Ampère. Then, we consider the minimal value imposed by the worst-case band-width, and signal-to-noise (SNR) ratio.

5.3.1. Band-width constraints on biasing current

We take as reference (for a fist approximation) the switched-capacitance non-inverting amplifier, which is well know from basic electronics, like from (Razavi 2001). Notice that, as explained for example from (Razavi 2001), in a simple amplifier, the first pole of the transfer function is defined by the output node small-signal resistance r_o and the target load capacitance $C_L = C_H$. For instance, the open loop bandwidth in Hertz is $B_w = (2\pi r_o C_L)^{-1}$, and the closed loop bandwidth (and the

respective time constant) can be obtained by the gain-bandwidth-product relation. Nevertheless, our circuitry considers switched-capacitor-amplifiers. Then, we have set our biasing current to meet the time constants for the equivalent circuits in both sampling and amplification modes. From general knowledge about electronics design, the settling time for the output signal (e.g. the time constant τ multiplied by 6) must be smaller than the clock semi-period. Then, for the sampling and amplification, respectively, we have the following relation:

$$6 \cdot \tau_{samp/amp} < \frac{T_{clk}}{2} \Rightarrow 6 \cdot \tau_{samp/amp} < \frac{1}{2 \cdot H \cdot fps}$$

Equation 28

Notice that $T_{clk} = (H \cdot fps)^{-1}$ since we assume that one row is processed during each clock period. This, however, is not correct when the several sampling stages are required, such as the circuit in Figure 42. In such case, a faster clock tick is required (for following the output speed of adjacent stages), and the fps is multiplied by a factor reflecting several sampling steps. We represented this factor as N_c. We assume that one stage can have a faster clock if required. From fundamental electronics, such as from (Razavi 2001), the equivalent sampling time constant is:

$$\tau_{sam} = \left(R_{on1} + \frac{1}{G_m}\right) C_H$$

Equation 29 : the time constant "τ_{sam}", for a detailed explanation, see (Razavi 2001)..

Where τ_{sam} is based on the equivalent circuit of the unity-gain sampler in sampling mode. In Equation 29, we neglect R_{on} as a first approximation. G_m corresponds to the input transistor transconductance, which we approximate as $G_m = g_m = 10 \cdot I_D$. Then,

$$\tau_{sam} = \frac{C_H}{10 I_D}$$

Equation 30

As explained in (Razavi 2001), for the amplification phase, the equivalent small signal circuit has a resulting time constant τ_{amp} as in Equation 31.

$$\tau_{amp} = \frac{C_L C_{eq} + C_L C_2 + C_{eq} C_2}{G_m C_2}$$

Equation 31 : time constant τ_{amp}, for a detailed explanation, see (Razavi 2001).

In Equation 31, C_{in} represents the operation amplifier input capacitance (Razavi 2001), which we neglect (compared to C_H) for a first approximation. For simplification, we assume that $C_L = C_2 = C_1 = C_H$. Then:

$$\tau_{amp} = \frac{3C_H}{g_m} \approx \frac{3C_H}{10 I_D} = 3\tau_{samp}$$

Equation 32

From last equation, we notice that the amplification phase is the critic one for selecting the biasing current. Then,

$$6\tau_{amp} < \frac{1}{N_c \cdot 2 \cdot H \cdot fps}$$

Equation 33

Where the factor N_c reflects that some stages have a faster clock. Also:

$$\frac{3C_H}{10I_D} < \frac{1}{N_c \cdot 12 \cdot H \cdot fps} \Rightarrow$$

$$I_D > 3{,}6C_H N_c H \cdot fps$$

Equation 34

Table 8 : example of values calculated for I_D when $C_H = 100\ fF, N_c = 1$ and $H = 500\ rows$.

fps	I_D (nA)
30	5,4000
60	10,8000
120	21,6000
300	54,0000
500	90,0000

From last table, we observe that the previous choice of $I_D = I_{amp} = 1\ \mu A$ was rather high in comparison with overall values calculated. For a typical smart imager, 500 pixel-rows and 60 fps is reasonable, leading to a bias current in the order of nA. Taking into account over-simplification made in this exercise, we can elevate again by one order of magnitude for a more conservative reference value. In last section, and in Table 7, we took the biasing current as μA. Now, we will exchange this by values found in Table 8, as presented in Table 9:

Table 9 : updated power current estimation when taking V_ampDD=1,5 V,C_H=100fF,H=W=500 pixels,V_R=1 V, and fps = 60.

Nc for each stage						Current (nA)					
Stage 1	Stage 2	Stage 3	Stage 4	Stage 5	Stage 6	Stage 1	Stage 2	Stage 3	Stage 4	Stage 5	Stage 6
1	9	1	2	1	1	10,80	97,20	10,80	21,60	10,80	10,80
1	9	1	2	1	1	10,80	97,20	10,80	21,60	10,80	10,80

From Table 9, we have enough information for calculating the total estimated power consumption. The calculation summary is presented in Table 10.

Table 10 : updated power estimation when taking V_DD=1,5 V,C_H=100fF,H=W=500 pixels, and fps = 60.

Power							Total power (nW)	Power per pixel (fJ/pix)
Alpha	Stage 1	Stage 2	Stage 3	Stage 4	Stage 5	Stage 6		
1	16,20	145,80	16,20	324,00	32,40	32,40	850,50	56,7
0,1	1,62	14,58	1,62	32,40	3,24	3,24	85,05	5,67

Next section will be dedicated to an analysis on noise, in order to assess if such multi-staged preprocessing is reasonable in terms of SNR degradation before the ADC.

5.4. Noise analysis

In this section, we study the impact of thermal noise on the SNR. The effect of flicker noise, fixed pattern noise, and charge injection in switched-capacitors can be added in further works. Here, we will take into account only the thermal component.

For a first approximation, we observe the spectral input-referred-noise (thermal) per unit of bandwidth (or PSD_{in}) intrinsic to a differential pair (based on (Razavi 2001)) with a capacitive load at the output C_H. Firstly, we take into account only the open loop case:

$$PSD_{in} = 8kT\left(\frac{2}{3g_m} + \frac{g_{ds}}{g_m^2}\right)$$

Equation 35 (Razavi 2001)

If the overdrive voltage $V_{ov} = V_{gs} - V_{th} = 0{,}2\ V$, then $g_m = 10I_D$. Also, $g_{ds} = \lambda I_D$. If $\lambda = 0{,}1\ V^{-1}$, then $g_{ds} = 0{,}1\ I_D = 0{,}01\ g_m$. Then,

$$PSD_{in} = 8kT\left(\frac{2}{3 \cdot 10I_D} + \frac{I_D}{10 \cdot (10I_D)^2}\right) \approx 8kT\left(\frac{2}{3 \cdot 10I_D}\right)$$

Equation 36

The noise-spectral-density PSD_{out} (thermal) is obtained from the product of PSD_{in} and the square of the transfer function magnitude $|H(f)|^2$. Firstly, we derive the output noise equation for the open loop case, and then we proceed for the closed loop case. For the open loop case, the gain $A = r_o g_m$, $B_w = (C_H r_o)^{-1} = \omega$ in radians per second. Notice that r_o is the open loop output resistance such that $r_o = r_{on} \,||\, r_{op}$ of the differential pair. From common knowledge, the transfer function is:

$$H(s) = \frac{V_{out}}{V_{in}} = A_0\left(\frac{1}{s\tau_0 + 1}\right)$$

Equation 37

$$|H(s=j\omega)|^2 = A_0^2\left(\frac{1}{4\pi^2 C_H^2 r_o^2 f^2 + 1}\right)$$

Equation 38

From common knowledge, we can make the replacement $x = 2\pi C_H r_o f$, with $f \in [0, \infty[$, then:

$$|H(s=j\omega)|^2 = A_0^2\left(\frac{1}{x^2+1}\right)$$

Equation 39

From common knowledge, the open loop output noise power is:

$$\sigma_0^2 = \int PSD_{in}|H|^2 df \ , \ df = \frac{dx}{2\pi C_H r_o}$$

Equation 40

Then,

$$\sigma_0^2 = PSD_{in}\int_0^\infty A_0^2\left(\frac{1}{x^2+1}\right)\cdot\frac{dx}{2\pi C_H r_o} = PSD_{in}\frac{A_0^2}{2\pi C_H r_o}\int_0^\infty\left(\frac{1}{x^2+1}\right)\cdot dx$$

$$\int_0^\infty\left(\frac{1}{x^2+1}\right)\cdot dx = \frac{\pi}{2} \Rightarrow$$

$$\sigma_0^2 = PSD_{in}\frac{A_0^2}{2\pi C_H r_o}\cdot\frac{\pi}{2} = \frac{PSD_{in}A_0^2}{4\tau_0}$$

Equation 41

With $\tau_0 = C_H r_o$[25]. We now take the closed loop case, in which the transfer function can be expressed as (Razavi 2001):

$$H_1(s) = \frac{V_{out}}{V_{in}} = \frac{\frac{A_0}{s\tau_0+1}}{1+\beta\frac{A_0}{s\tau_0+1}}$$

Equation 42 (Razavi 2001)

With β begin the feedback gain. Last equation can be re-arranged as (Razavi 2001):

$$H_1(s) = \frac{\frac{A_0}{1+\beta A_0}}{1+\frac{s\tau_0}{1+\beta A_0}}$$

Equation 43 (Razavi 2001)

We can now make some convenient replacements:

[25]In (Razavi 2001), page 252, they use ω_0 in Equation 42 and Equation 43 instead of τ_0. Here, we take $\tau_i = \omega_i^{-1}$

$$A_1 = \frac{A_0}{1 + \beta A_0}$$

Equation 44

$$\tau_1 = \frac{\tau_0}{1 + \beta A_0}$$

Equation 45

From equations above, the so-called (in the literature) gain-bandwidth product GBW can be derived:

$$GBW = \frac{A_1}{\tau_1} = \frac{A_0}{\tau_0}$$

Equation 46

Then, the closed loop transfer function H_1 becomes:

$$H_1(s) = \frac{V_{out}}{V_{in}} = A_1\left(\frac{1}{s\tau_1 + 1}\right)$$

Equation 47

By comparing with equations from the open loop case, we can immediately conclude that:

$$\sigma^2 = PSD_{in}\left(\frac{A_1^2}{4\tau_1}\right) = \frac{PSD_{in}\left(\frac{A_0}{1 + \beta A_0}\right)^2}{4\left(\frac{\tau_0}{1 + \beta A_0}\right)} = PSD_{in}\left(\frac{A_0^2}{4\tau_0}\right)\left(\frac{1}{1 + \beta A_0}\right)$$

Equation 48

If we take a simply unity gain buffer as reference, then $\beta = 1$, and:

$$\sigma_1^2 = \frac{PSD_{in}A_0}{4\tau_0} = PSD_{in} \cdot \frac{r_o g_m}{4C_H r_o} = PSD_{in} \cdot \frac{g_m}{4C_H} \approx 8kT\left(\frac{2}{3 \cdot 10I_D}\right)\left(\frac{10I_D}{4C_H}\right)$$

$$\sigma_1^2 \approx \frac{4kT}{3C_H}$$

Equation 49

The effective number of bits ENOB is (Maloberti 2007):

$$ENOB = \frac{SNR_T - 1.78}{6{,}02}$$

Equation 50 (Maloberti 2007)

Where SNR_T is the signal to total-noise ratio (not only quantization noise) (Maloberti 2007). The typical formula in the literature for the signal to noise ratio is:

$$SNR = 20 \cdot \log\left(\frac{V_{sig}}{\sqrt{\sigma_1^2}}\right)$$

Equation 51

Some example calculations are presented in Figure 50 and Figure 51, where σ^2 is the calculated noise-power σ_1^2 multiplied by the "equivalent number of sequential amplifications n":

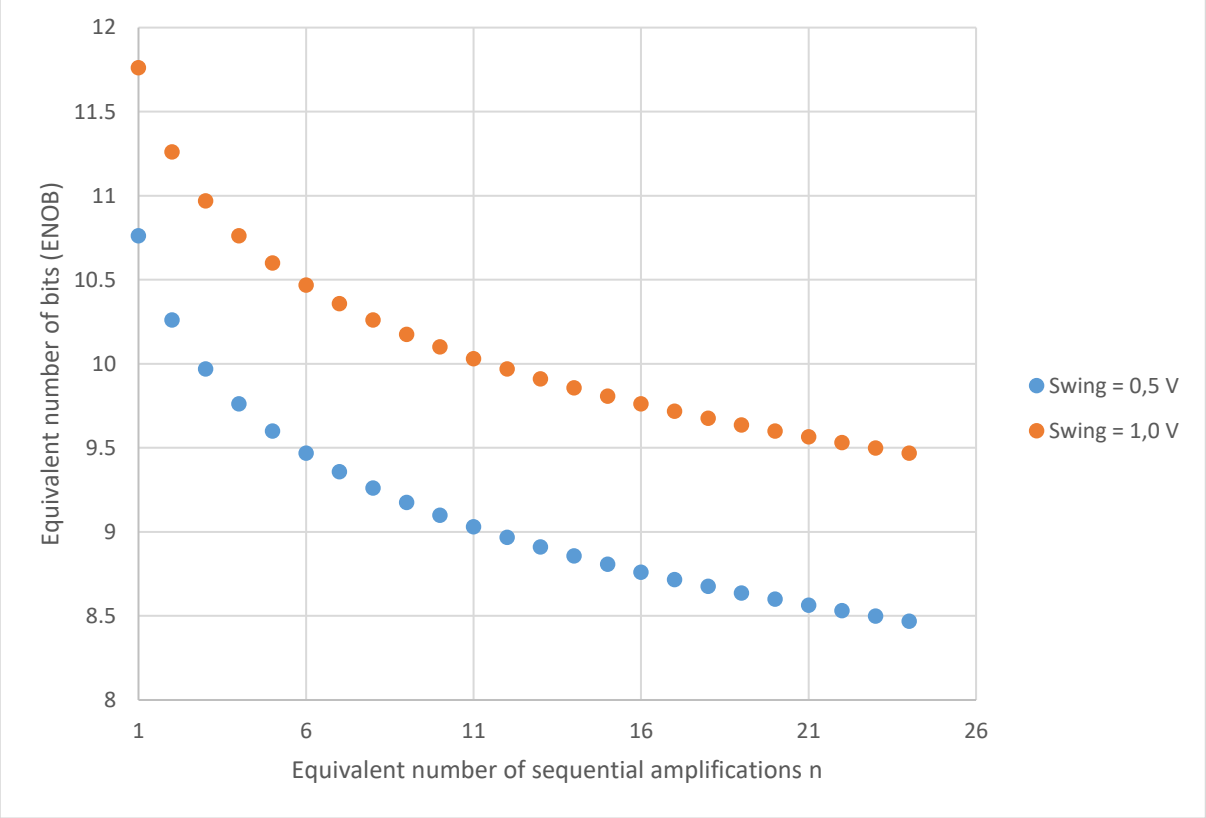


Figure 50 : Equivalent number of bits for two different pixel-intensity-voltage-swings V_sig=V_R, as a function of the equivalent number of sequential signal amplifications n. C_H=100 fF,T=300 K.

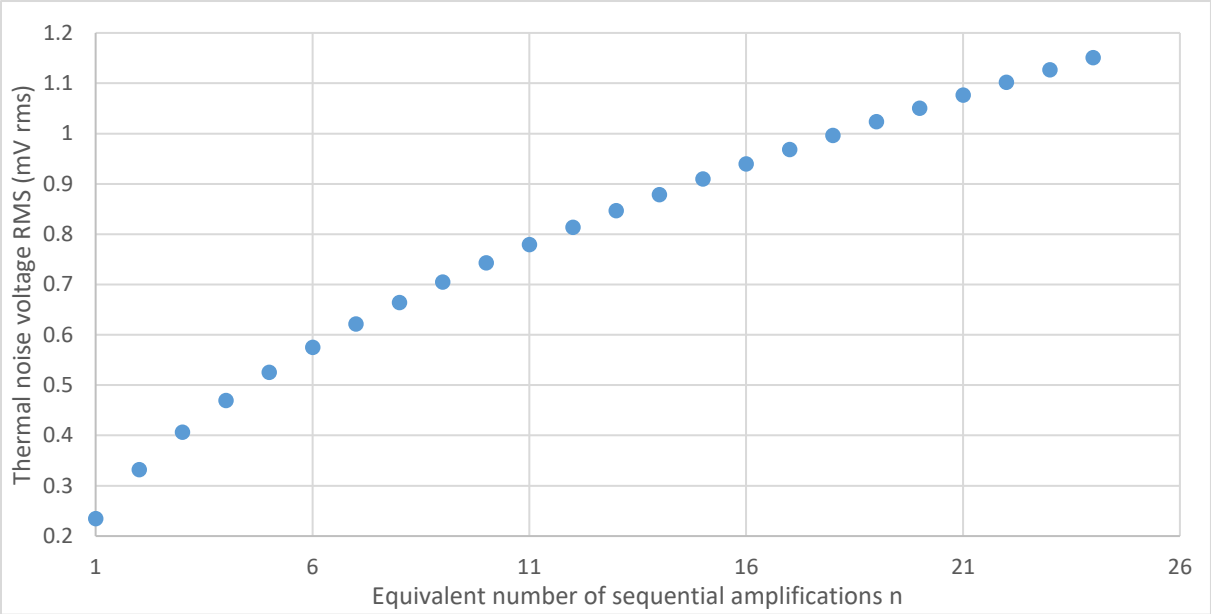


Figure 51 : RMS Thermal noise voltage (√(σ^2)), as a function of the equivalent number of sequential signal amplifications n. C_H=100 fF,T=300 K.

This number of amplifications n was inferred from circuits presented from stages 1 to 5. For our focus pipeline, $n = 24$, if we assume that all amplifications had a noise value close to the noise value of a closed-loop gain of 1. For the case of the subtraction and addition operations in stages 4 and 5, the noise propagates in two branches before those two branches are added again for performing the operation. That is why $n = 24$ is higher than the total number of amplifiers in any non-cyclic sequential path from CDS to quantization. Then, Figure 50 and Figure 51 illustrate how the signal degrades due to thermal noise through the sequential amplifications. We also evaluate the ENOB for two different values of V_R, 0.5V and 1V, in order to a reasonable voltage range for biasing the pipeline. We can guess that this range could be divided by 2 at the end of the pre-processing, meaning that the area between the two curves in Figure 50 is the expected calculated interval. We observe that ENOB drops from 10,76 (n = 1) to 8,47 (n = 24) if the useful swing is 0,5 V. If the useful swing is 1 V, then ENOB drops from 11,76 to 9,47. Then, for n = 24, from previous estimations the resulting ENOB should be among 9,47 and 8,47. If we further consider 1/f noise and switches charge injection, we can just keep in mind that our analog pipeline can be approximated to an 8-bit digital pipeline.

From last paragraph we infer that the noise-power only due to thermal power from the multi-staged analog pre-processing has an significant contribution. Nevertheless, here is when one of the "powerful" characteristic of oriented gradients as "low level static features" comes in: since those correspond to a relatively high local and spatial gradient, then they are more robust to noise if the expected gradient variation is significantly higher than noise signal in rms. In addition, we can compare the worst estimated ENOB of 8,47 with the amount of bits codifying the gradient magnitude (1 bit) or the gradient angle (2 bits). Moreover, this features still allow to perform low power object detection.

5.5. Conclusions of chapter 5

In this chapter, we explained and analyzed the circuit stages to achieve the pre-processing pipeline mentioned in chapter 4. We differ from (Young et al. 2019; Soell et al. 2016) as we introduced the average denoising stage in order to alleviate the memory and runtime of the next processing stage, the Edge Boxes algorithm. We obtained first estimations of power consumption, and we refined such estimations by taking into account the required biasing current for the desired bandwidth. This lead to an ideal power consumption of ~851 nW (at 500 x 500 resolution, 60 fps and without duty-cycling the amplifiers), corresponding to a FOM of **56.7 fJ/frame/pix**. We also discussed that duty-cycling with a low value of α is especially interesting when the smart imager "decides" to reduce the frame rate in order to save power.

Next, we made an analysis on the impact of thermal noise in the final gradient computation. It showed that our analog pipeline can be approximated to an 8-bit digital pipeline. This is a critical analysis since a complex analog pre-processing architecture with several sequential stages might significantly degrade the output signal. We observed that indeed that is the case, obtaining a drop 11,76 - 10,76 to 9,47 - 8,47 equivalent bits. However, even though this could be critical for standard CMOS imagers, typically aiming for high quality images, in chapter 6 we show that this does not prevent our proposed pipeline from functioning, even if the output gradient magnitude is codified in 1 bit, and the angle in 2 bits.

In next chapter, we will benchmark different variations of edge-extractors architectures already detailed in the two previous chapters. We will focus on a specific dataset so we can quantify performance, mainly in terms of runtime, memory usage and Intersection-Over-Union.

Chapter 6. Object Localization benchmarks

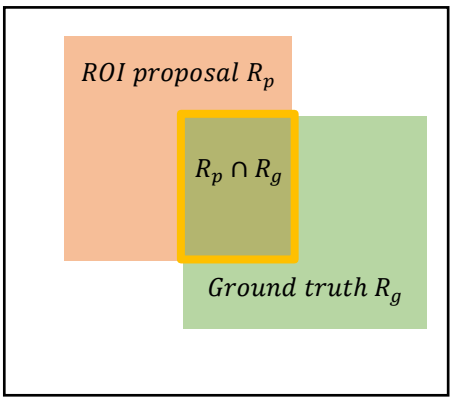
In this chapter, we try to assess the optimal pre-processing architecture by characterizing the system performance for object localization. As mentioned in chapter 4, a variety of object detection (OD) algorithms behaves as a two-staged process: firstly, the localization of potential objects in the scene, and secondly, the classification of the content inside each object proposal or region of interest "ROI". Moreover, each task (localization or classification) performs as a "black box", and their contents can change as long as they maintain the expected input and output variables.

6.1. Localization characterization methodology

When classification and localization are indeed two separate stages, a metric for each stage can be relevant and simpler to carry out. For the localization phase, the "intersection over union" (IoU) or "Jaccard index" ("Jaccard Index" 2021) is typically used in the literature. Conceptually, it measures how well an individual ROI proposal overlaps with a ground-truth-bounding-box thanks to a scalar between 0 (no overlap at all) to 1 (perfect overlap). Mathematically, we can express it by considering two rectangular areas in the image: one corresponding to a single ROI proposal $\boldsymbol{R_p}$, and another one corresponding to the ground-truth-bounding-box for a particular object $\boldsymbol{R_g}$ (both represented in Figure 52). Then,

$$IoU(R_p, R_g) = \frac{R_p \cap R_g}{R_p \cup R_g}$$

Equation 52 ("Jaccard Index" 2021)



$R_p \cap R_g$: intersection of two areas

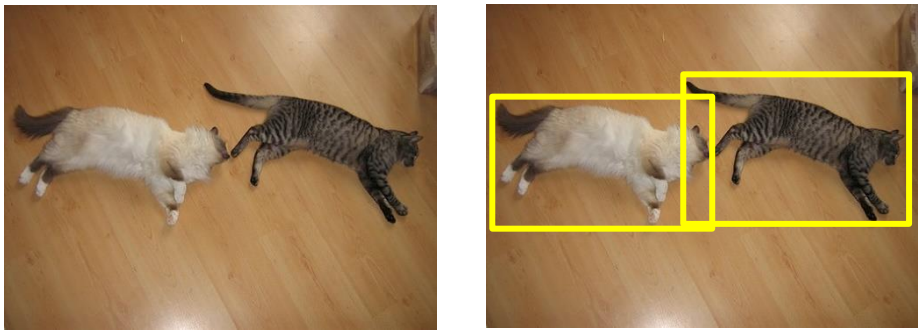
$R_p \cup R_g$: union of the two areas

Figure 52 : Illustration of the IoU showing the ROI proposal, ground-truth-box, and intersection areas. The union area is delimited by borders of both rectangles without taking into account borders of the intersection rectangle.

Typically, dataset annotations provide ground-truth-boxes (in plural, if there are several objects in the image), whereas a computer algorithm gives the ROI proposals. Notice that only one proposal may be insufficient, thus the algorithm may output a certain number of them. For instance, Edge-Boxes has a parameter controlling the amount of output boxes. In this work, we fix it to 1000 boxes in all benchmarks, since it was a value used by (Zitnick and Dollár 2014). If the reader wants to go deeper in the effect of the amount of boxes, the Edge-Boxes paper (Zitnick and Dollár 2014) explains the trade-off between IoU and the amount of proposals.

In order to benchmark different architectures in term of IoU, we require a dataset compatible with OD. We selected the dataset Pascal VOC 2007 (M. Everingham et al. n.d.; n.d.; Mark Everingham et al. 2015) since it is a well-known dataset, with illustrative classes of objects (person, cat, dog, car, among others). Indeed, it has 20 classes in total (see table Table 11). The dataset has a set of images, and for each image, there is a file containing the annotations. Inside that file, one can find, for each object, a category (corresponding to one of each class), and image coordinates for the ground-truth object bounding-box. Table 11 presents a summary of the classes provided by this dataset. In next section, we start by explaining how we use our simulation framework EdgeTon (described in chapter 3) to characterize different architectures followed by the Edge-Boxes algorithm.

Table 11 : classes found in the Pascal VOC 2007 dataset(M. Everingham et al. n.d.; n.d.; Mark Everingham et al. 2015), and one example image with two cats.

(a) Classes	(b) Example
aeroplane	 Image 000019.jpg in the PASCAL VOC 2007 dataset (M. Everingham et al. n.d.; n.d.; Mark Everingham et al. 2015), the image contains two objects, both of category 'cat', and the bounding boxes for each are given in a XML file corresponding to this image. On the right, we provide the original image. On the left, we draw (inaccurately) the bounding boxes for an illustrative purpose.
bicycle	
bird	
boat	
bottle	
bus	
car	
cat	
chair	
cow	
diningtable	
dog	
horse	
motorbike	
person	
pottedplant	
sheep	
sofa	
train	
tvmonitor	

In the case of Edge-Boxes, we do not train the algorithm, and we just keep the parameters by default, and with the amount of proposals set to 1000. Then, for localization benchmarking with Edge-Boxes, we only use the training portion of the dataset[26]. Next, we must take into account that along the training-dataset there are images with several objects of different categories, and so we proceed as follows:

[26] For readers that are not familiar with the term « training dataset », we clarify that for training machine learning algorithms (e.g. with parameters that are learned), the dataset is typically split in three portions: firstly, the training dataset, which is used to learn the parameters (for instance, by means of an optimization algorithm). Secondly, the validation dataset, which is used to tune algorithm parameters that are not directly learned (e.g. the number of layers in a CNN, regularization parameters, the learning rate, etc.). Finally, there is the test dataset, which is used to characterize the performance of the already trained model with samples that haven't been « seen » before.

6.1.1. Simulation Flow with EdgeTon

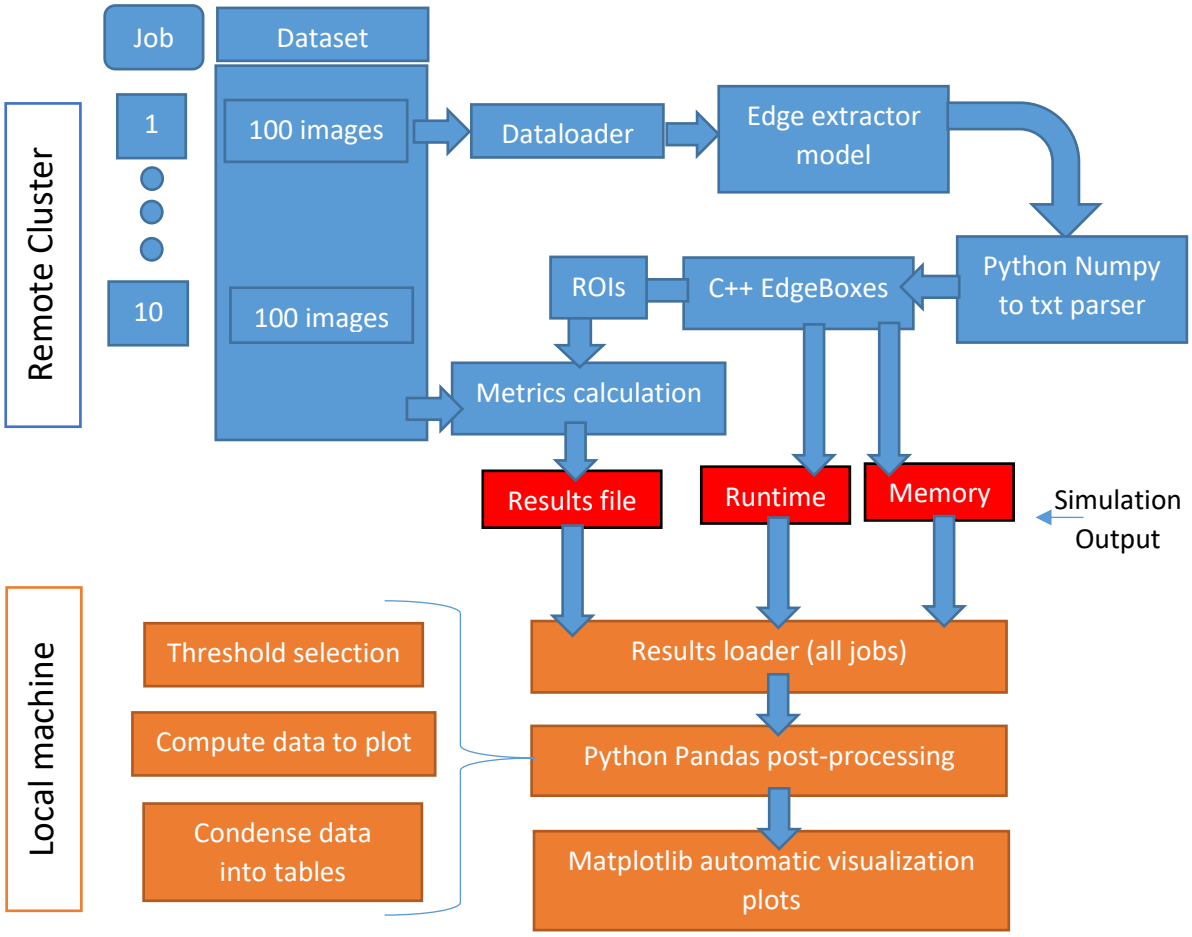


Figure 53 : Diagram of simulation flow from dataset to final plots. The figure is divided in upper and bottom sections. The first one was computed in a cluster, with 10 jobs (available to run in parallel and scalable to more jobs) of 100 images each. The second (bottom) one corresponded to the post-processing made to generate the final plots.

Figure 53 shows the overall simulation flow from dataset to final plots. The simulation was made such that several metrics could be obtained: firstly, the IoU for localization. Secondly, variables showing (indirectly) the relative time and space complexity of Edge-Boxes when using different extractors. All stages show in previous figure, except for the Edge-Boxes source code (Dollar and Zitnick 2015b), which was not changed, were coded in this work (on top of known libraries such as OpenCV, Numpy, Pandas, Matploblib, etc). All stages regarding the cluster (including the management of parallelization of batches of images) are a part of a bigger simulation scheme comprised in EdgeTon. The simulation is scalable to more jobs if desired, yet we arbitrarily chose 10 jobs, each containing 100 images chosen each time randomly from the original dataset.

Notice that EdgeTon runs in Python code, yet the source code for EdgeBoxes (Dollar and Zitnick 2015b) is provided in C++ code. Since we did not want to change it, we took the source code, from which a main class for generating ROI proposals was defined. We instantiated this class inside another script for loading the parsed images from the .txt files and wrapping with the rest of the simulation (i.e. to output files in the desired path and format), and for counting the approximated runtime of

EdgeBoxes with C++ std::chrono library. In order to approximate the memory usage, we focused on obtaining two variables that we found to be most relevant for comparisons: the segment-affinities variable size, and the amount of segments clustered by Edge-Boxes. In order to access them, we modified the source code (Dollar and Zitnick 2015b) with functions that allow to output in a file the values of those variables during runtime. Then, we had to slightly change the source code, but such changes do not affect the behavior of Edge-Boxes, except for the overhead to write the output ROI proposals, runtime, segment-affinities size, number of segments, number of columns and number of rows per image.

In next sections, we describe our simulation flow with more details.

6.1.2. Edge extraction with a smart imager behavioral model

For this stage, we use our framework EdgeTon to simulate an approximation of different smart imager architectures. Here, the overall input is an image from the dataset, and the output is the edge map containing magnitudes and orientations. The way those imagers are modeled is described in chapter 3, where the idea is to model different kinds of oriented gradients (or edge-map) extractors for integrated/embedded smart imagers. The types of edge extractors we took into account are summarized below in Table 12.

Table 12 : different types of light-intensity oriented gradient extractors types considered in our localization benchmark.

Oriented Gradient extractor type	Description	Output to EdgeBoxes backend bit-depth	Threshold type
Linear (Digital)	Uses kernels in Equation 17 and the digital model described in chapter 3.	1 bit for magnitude 2 bits for orientation	Fixed (4-bit digital)
Linear (Analog)	Uses kernels in Equation 17 and the linear analog model described in chapter 3.	1 bit for magnitude 2 bits for orientation	Fixed (32 bit-analog)
Relative	Uses kernels in Equation 17 with dynamic range enhancement described in chapter 4, and the analog model described in chapter 3.	1 bit for magnitude 2 bits for orientation	Light intensity dependent, with a fixed factor
Logarithmic	Uses logarithmic kernels as in (Omid-Zohoor et al. 2018; Young et al. 2019), and is modeled as described in chapter 3.	1 bit for magnitude 2 bits for orientation	Light intensity dependent, with a fixed factor
8-bit Sobel	Uses the Sobel kernel as in Equation 16, and outputs 8-bit (ENOB) oriented edges in floating point representation map as described in chapter 3.	8 bits (ENOB) for magnitude and orientation (32-bit float representation)	Fixed (32-bits)

Moreover, for each extractor type we take into consideration two variants, namely if a blur kernel is applied before the intensity derivative (or gradient) extraction (with one from previous table), or no blur-step at all. In addition, we study the effect of 2x2 binning for an illustrative purpose. Once the output is obtained, it is passed to next stage: Edge-Boxes. One important aspect is that each extractor type has one parameter that we did not fixed, but that we optimized from simulations: the threshold. Indeed, whether it is constant, or proportional (factor) to a local average (for logarithmic and relative types), we did not have any particular design criteria to choose them. Then, we performed a parameter sweep for each architecture and chose a specific threshold (factor) to report our results.

In order to do so, we obtain the localization performance for all extractor types and several variants (regarding blur kernel and binning). We obtained results for several thresholds (factors) values and for 10 jobs (iterations). Each iteration could give different values of the optimal threshold, since they used different sub-sets of 100 images. Then, we chose the threshold that repeated the most from all jobs. Next, for reporting benchmark results, we only use the threshold that was chosen, regardless if it was optimal or not for any particular job.

6.1.3. Localization with Edge-Boxes

We now feed the edges-maps from previous section to a compiled binary containing the source code for Edge-Boxes from (Dollar and Zitnick 2015b), and some wrapping code. For each image, the corresponding edge map is loaded. Since quantization was already taken into account, the pixel values (both for gradient and magnitude) can be converted to floating point or integer C++ types and quantization is still preserved. This is analogue to saying that the smart imager output has been passed to a dedicated computed hardware running on 32-bit floating point resolution.

The localization stage output a file per image, containing 1000 ROI proposals coordinates and their associated objectness scores. Moreover, another file (one per job) contains the computational complexity required for those 1000 ROI proposals: the runtime for each image, the size (in number of elements) of the segments-affinities variable, the number of segments clustered by Edge-Boxes, and the number of columns and rows for each image for normalization (so the runtime and memory estimation do not depend on image resolution).

6.1.4. Intersection over union metrics calculation

We already introduced the IoU as a metric for characterizing performance for object localization. Now, we go deeper into how we used it in conjunction with the Pascal VOC 2007 dataset. Notice that IoU simply gives a relation between a single ROI proposal and a single object ground-truth-box. However, there are 1000 ROI proposals per image, potentially several objects per image, and 20 different classes of objects. One problem we faced is that a single object in the image can be "proposed" several times with different boxes from Edge-Boxes. This can happen because the algorithm distributes hypothetical boxes of different sizes and aspect ratios along the image, and it is possible that two near boxes (that were similar enough) both score high in objectness. Then, one question rises: which of all of those potential boxes do we take for calculating the IoU with the ground-truth-box? In this work, we took as reference the "best IoU" or BIoU, which we will further develop in next paragraph.

For BIoU calculation of a single object of a single image, we considered boxes that had an IoU equal or greater than 0.5, called R_{pk} , with a particular ground-truth-box R_g. Then we selected, among those R_{pk}, the one with the best IoU score, noted R_{pz} . In other words R_{pz} verifies $IoU(R_{pz}, R_g) > IoU(R_{pk}, R_g)$, $R_{pk} \neq R_{pz}$. Rejected R_{pk} are considered as false positive. Another interesting case is if there is no ROI proposal R_p such that $(R_p, R_g) \geq 0{,}5$, in which case R_g was counted as a "false negative". Thus, for every R_g , 4 pieces of information are returned: a specific bounding box R_p, a related IoU score (BIoU) (it could be zero in case of no overlapping), a number of false positives and a number of false negatives. The performance of object localization can thus be assessed independently

of the classifier, by assuming that the best chance that the algorithm (classifier) had of classifying properly the object inside the proposal, was for the R_p related to the BIoU.

Another relevant aspect is that, after processing all images, our metrics tool is able to calculate a metric per class. Indeed, even though the localization with Edge-Boxes is class agnostic, there was a change that some types of objects could score higher in objectness respect to others. Another reason for spreading performance results into classes is to avoid biasing results if the dataset is "unbalanced": if there are much more samples of one class respect to another, then the average along one class will have a greater impact on the final performance score[27]. For instance, suppose we have a dataset with only persons and pets, and which contains 1000 samples of persons and 10 samples of pets. Then, assume that the average best IoU, referred to as ABIoU, for all persons was $0{,}5$ and 0.1 for pets. The global average is:

$$ABIoU(all\ classes) = \frac{0{,}5 * 1000 + 0{,}1 * 10}{10 + 1000} = 0{,}49604$$

The last calculation reflects that even though the performance for pets localization was very bad (0,1), the ABIoU for all classes was minimally impacted. In order to have a global metric independent of the class sizes, we take first the ABIoU for all samples of the same class, then, the global ABIoU is the average of the averages per class. Coming back to the case of the hypothetical persons and pets dataset, we would have:

$$ABIoU(persons) = 0{,}5$$

$$ABIoU(pets) = 0{,}1$$

$$n_{classes} = 2$$

$$global\ ABIoU = \frac{\sum ABIoU_i}{n_{classes}} = \frac{0{,}5 + 0{,}1}{2} = 0{,}3$$

Equation 53

In last calculation, we can observe that now the fact that pets were badly localized is better reflected. In our benchmarks, we use the global ABIoU as metric for each behavioral model.

6.2. Benchmarks

6.2.1 Global ABIoU along jobs for each architecture

Our benchmark is illustrated in Figure 54 with a bar plot summarizing results for the average (along jobs) global ABIoU for several edge extraction variants. Figure 54 presents the average global ABIoU obtained with 10 jobs, each job processing 100 randomly selected images. Bar labels indicate

[27] Notice that our objective with the tool we developed was to use and charaterize (fastly) other datasets aside from Pascal VOC 2007 in future works.

the edge extractor type and variants (BINN refers to 4x4 binning). The red bars indicate the Average ABIoU for each (notice the range in the x axis not starting at zero), and the black lines indicate the standard deviation along the 10 jobs.

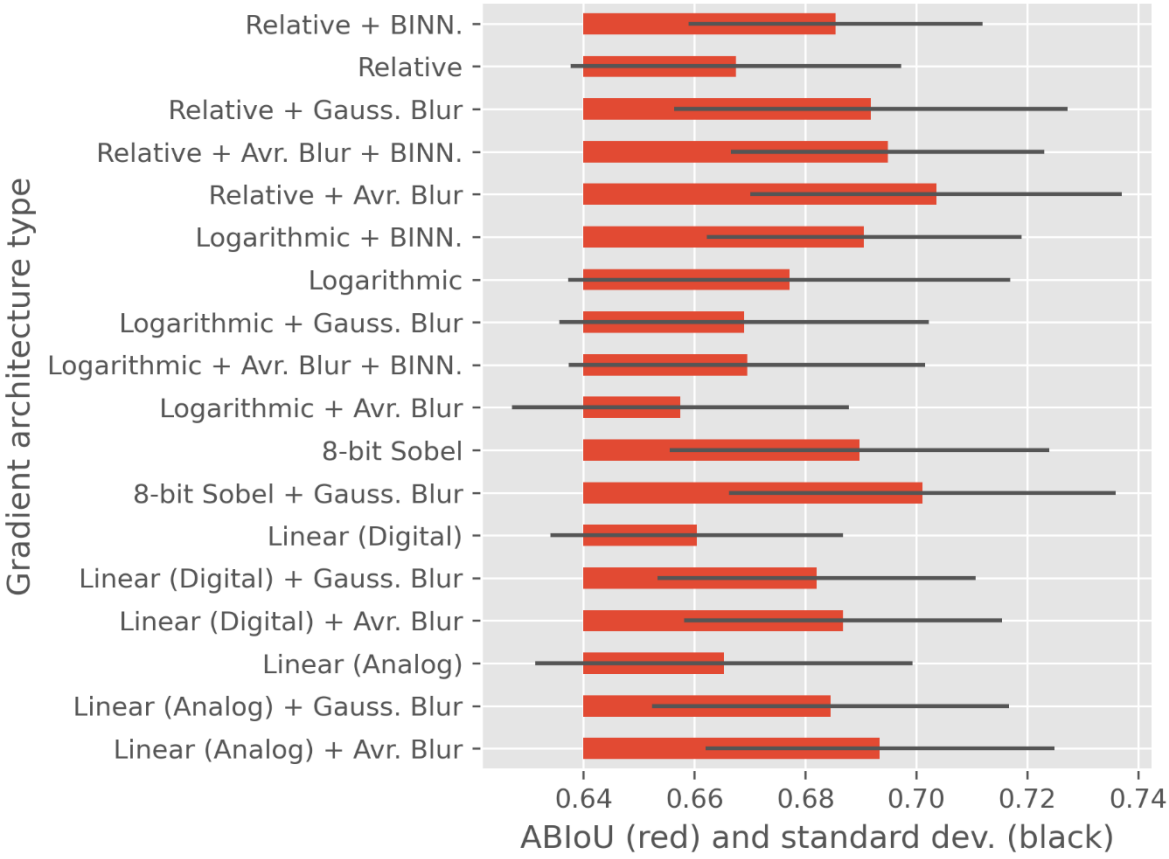


Figure 54 : average global ABIoU obtained with 10 jobs.

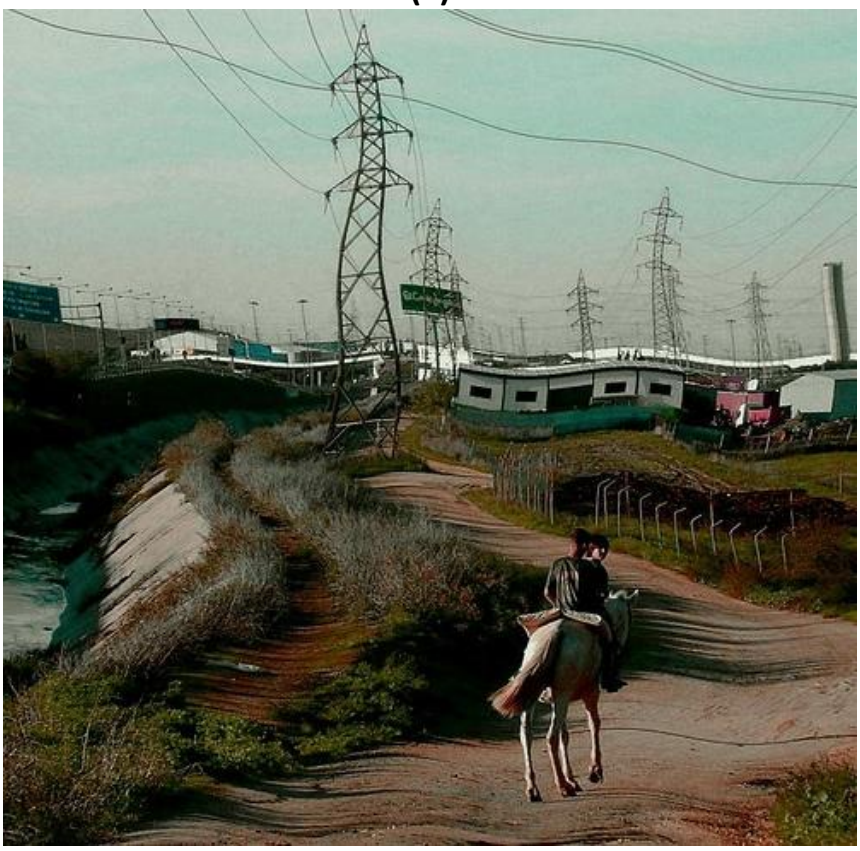
In addition, in Table 13, we summarize the selected thresholds for each architecture (with the method mentioned previously):

Table 13 : selected thresholds values issued from simulation (selected as described in chapter 3).

archName	selectedThres
analogOrigrad_averageBlur	30
analogOrigrad_gaussianBlur	30
analogOrigrad_noBlur	30
digitalOrigrad_averageBlur	30
digitalOrigrad_gaussianBlur	30
digitalOrigrad_noBlur	30
relativeOrigrad_averageBlur	0,3
relativeOrigrad_averageBlur_withBinning	0,3
relativeOrigrad_gaussianBlur	0,3
relativeOrigrad_noBlur	0,3
relativeOrigrad_noBlur_withBinning	0,6
logHog_averageBlur	1,8
logHog_averageBlur_withBinning	1,8
logHog_gaussianBlur	1,8
logHog_noBlur	1,8
logHog_noBlur_withBinning	1,8
highResOrigrad_gaussianBlur_sobelGradient	50
highResOrigrad_noBlur_sobelGradient	60

From Figure 54, we observe that, for all architectures, the global ABIoU (from here, we call it simply the ABIoU) was higher than 0.5, which is typically the minimum expected for “sufficient” ROI proposal (Zitnick and Dollár 2014). Yet, (Zitnick and Dollár 2014) suggests that this values may not be good enough, and proposes targeting higher IoUs. For instance, they (Zitnick and Dollár 2014) targeted IoUs of 0.5, 0.7 and 0.9 (which can be tunned with Edge-Boxes parameters). In this work, we take intermediate values from ~0.5-0.7 as target for Edge (Boxes parameters and we hypothesize without further demonstration that this is good enough for embedded applications. Of course, we observe the impact of gradients quantization falling down from 0.7 especially for the linear and logarithmic cases. We also observe that best performances come from the 8-bit Sobel case and the relative gradients. ROI proposals can face “difficult cases”, due to small, partially-occluded, or surrounded by fine-textures objects (such as grass). An image example is given on Figure *55* (a). The resulting Edge map and ROI proposal are on Figure *55* (b): the first two architectures give good proposals for one person and the horse, whereas they missed the second person that is partially occluded. In this case, the relative pre-processing failed to give a good proposal for both persons and only gave a good one for the horse.

(a)



(b)

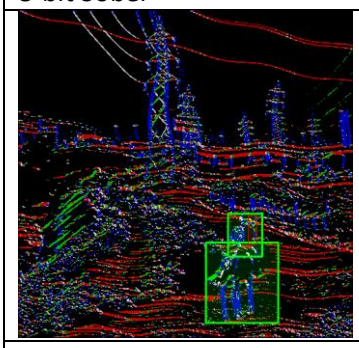
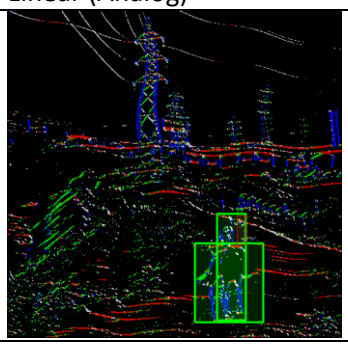
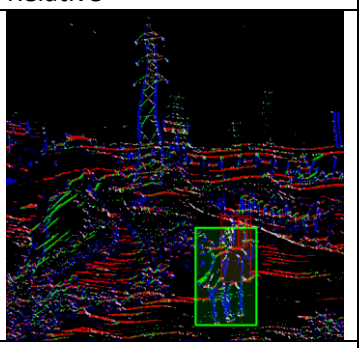
8-bit Sobel	Linear (Analog)	Relative
		

Figure 55 : example of an input image from *PASCAL VOC2007 dataset (M. Everingham et al. n.d.; n.d.; Mark Everingham et al. 2015), and corresponding representations of the oriented edges map outputs from three different edge-extractors. The gray/red boxes represent the best ROI proposals after running Edge-Boxes on the edges-maps. Green means that the object was correctly including in the ROI, while red means that the object was not found (e.g. all ROIs where such that their IoU < 0.5).*

We now focus in answering four key questions:

1. Is the average blur necessary or critical?
2. What is the impact of a 4x4 pixels binning?
3. Could an analog edge extractor have better performance than a similar complexity digital one?
4. What is the performance of the best possible analog extractor in comparison with a digital architecture running with 8-bit depth computations?

In the next paragraphs we will cover those questions from the point of view of ABIoU, memory and runtime. Finally, we will provide a FOM for estimating the optimal architecture in terms of those three metrics.

Blur vs. no blur impact on ABIoU

Figure 56 shows that, for most of the cases, the average blur improved the ABIoU. For readers that are not familiar with this kind of plot (box plot ("Box Plot" 2021)), the rectangles englobe the percentiles 25, the vertical smaller bars englobe the percentile 50, and values outside are represented with dots (considered as outliers). The red bar indicates the median value. Logarithmic gradients gave a lower performance respect to the relative ones, one reason could be a deteriorated angular approximation from the logarithmic gradient components. Indeed, during our experiments, we observed that edge maps for logarithmic gradient contained low occurrences of 45° and 135° edge orientations. The majority of detected edge-angles were 0 degrees (Gx = 1 or -1, Gy = 0) or 90 degrees (Gx = 0, Gy = 1 or -1). Thus, we think that this makes harder for Edge-Boxes to link different edges into segments and different segments into connected contours. We observed that, except for the case of logarithmic gradients, the average filter (blur) positively affects the ABIoU. Our explanation of why this happens (for the increasing cases), is that the blur step reduces (statistically speaking) the amount of false edges, which makes it easier for Edge-Boxes to generate meaningful segments and connected contours. In the case of the logarithmic gradients, this reduction may be too drastic, having then a negative impact.

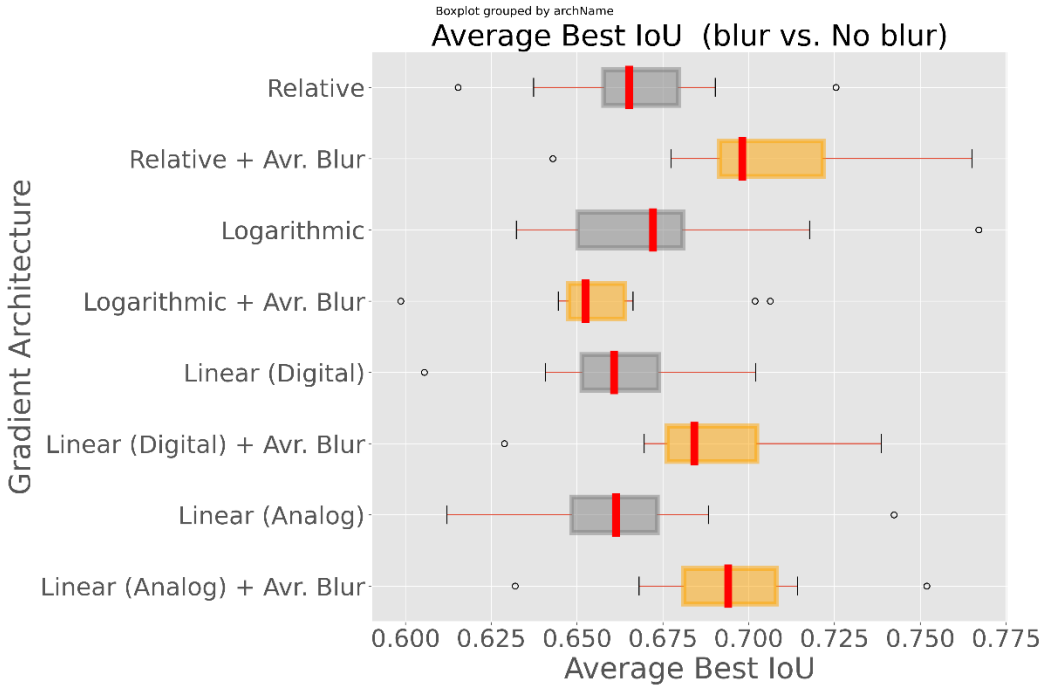


Figure 56 : global average best IoU comparison for architectures using and not using de-noising.

Binning vs. no binning impact on ABIoU

From Figure 57 we observed that binning has different impacts on relative and logarithmic gradients. Binning consistently improves the performance for logarithmic gradients, whereas the average denoising tends to have a negative impact. For relative gradients, both Binning and denoising have a positive impact. However, the best case scenario for relative gradients was when only the average denoising is applied. Our explanation of why the logarithmic gradients benefit of binning (in the contrary with relative gradients) is that Binning has two different impacts on the image. On one side, it reduces the amount of false edges, and the 2x2 averaging contributes in reducing false edges (yet it can give place to bounding effects since the averaging zones do not overlap). On the other side, it reduces the image resolution, thus neglecting small details which could have been used to recover the overall contour topology of any object. The first impact is positive, and the second is negative. We then think that the noise reduction due to binning is beneficial for logarithmic gradients in terms of the ABIoU. Notice that for the case of relative gradients, if the average blur is applied then binning decreased slightly the performance, meaning that in that case the noise reduction due to binning was less important than details lost due to the loss in image resolution.

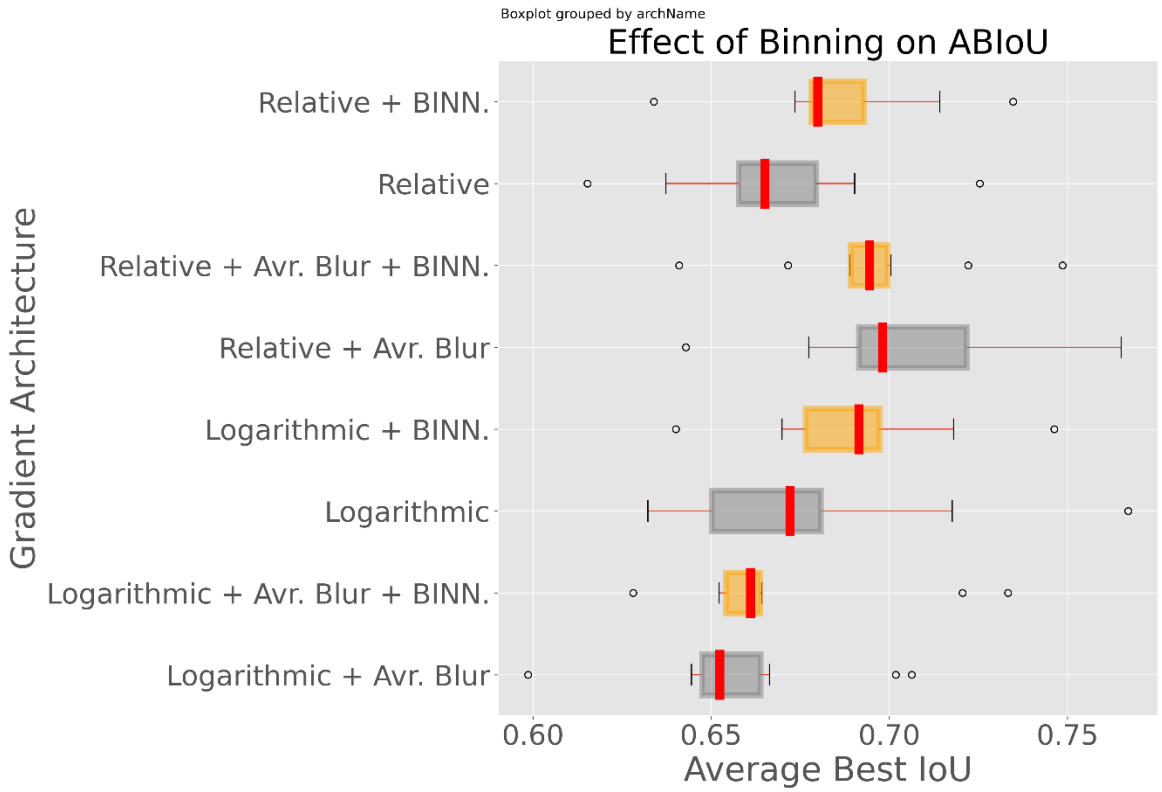


Figure 57 : global average best IoU comparison for architectures using and not using 2x2 binning.

Relative vs 4-bit linear gradients

Figure 58 shows results when comparing relative and simple linear gradients. As explained in chapter 3, relative gradients were calculated as follows: firstly, 8-bit input images were pre-processed in floating point resolution. Then, a first quantization to 4 bits (ENOB) was introduced, as described in chapter 3, before the final quantization to 3 bit edge maps was obtained. For the case of linear-analog

gradients, the same was performed, but with equation related to the simple linear edge-extractor. For the case of the linear-digital edge-extractor, the 8-bit image was first quantized into 4 bits, and then all operations until right before generating the 3-bit edge map are done in 4 bits. We observe from Figure 57 that relative gradients have a tendency towards greater values when an average blur is applied. Nevertheless, it is expected that there is no exceptional enhancement respect to linear gradients, since the PASCAL VOC 2007 dataset is not particularly made for testing high dynamic range image with very different illumination conditions along the same image.

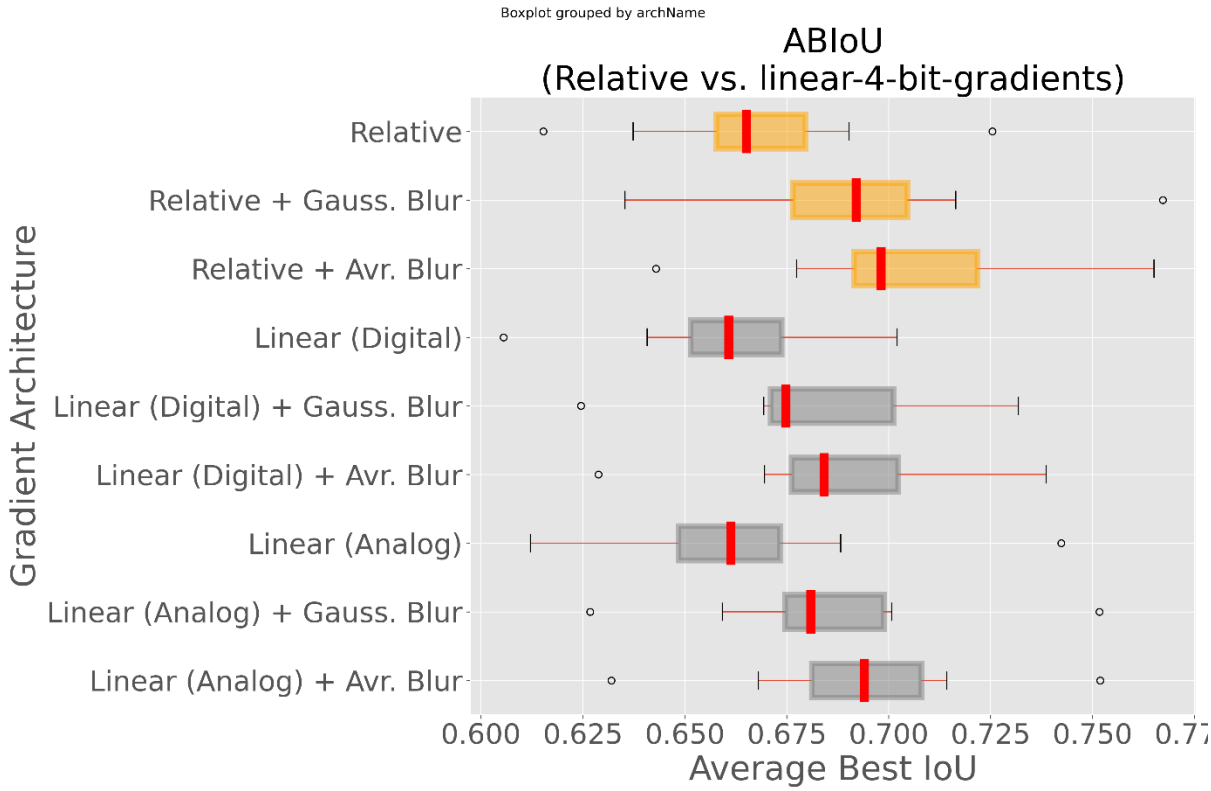


Figure 58 : global average best IoU comparison for architectures using relative and simple linear gradients (4 ENOB).

Relative vs 8-bit linear gradients

As a recall, 8-bit-linear gradients were calculated as follows: they were calculated similarly to the simple linear analog edge-extractor, with the difference that the Sobel kernel was used for the derivative. Moreover, right before generating the 3-bit edge map, a quantization noise with an ENOB of 8 bits was introduced instead of 4 bits (as explained in chapter 3). In Figure 59, we observed that the effect of the average filter (blur) was more important when the number of bits was lower (like for the relative case). However, when the average blur was applied, the performance of the relative gradients was as good as the equivalent 8-bit-Sobel architecture.

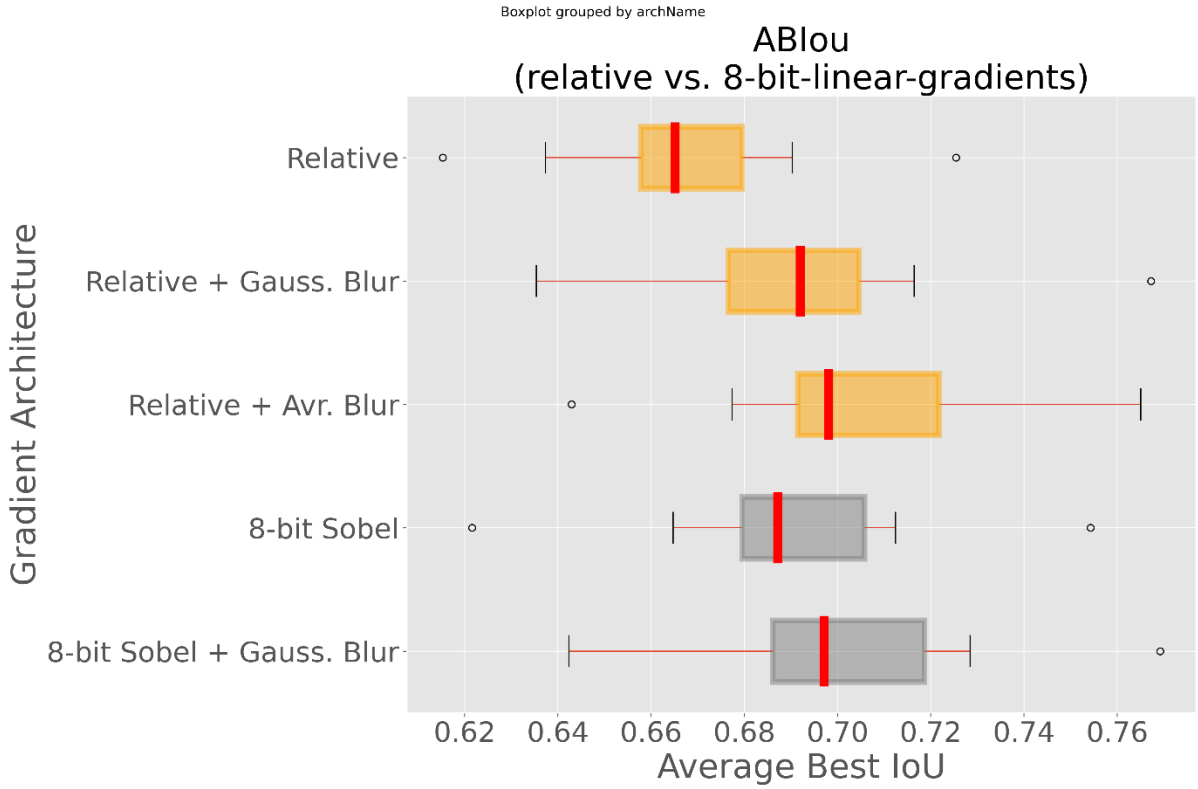


Figure 59 : global average best IoU comparison for architectures using relative and simple linear gradients (4 ENOB).

6.2.2 Memory estimation

After our analysis in chapter 4, we found that (theoretically) the affinities variable is the one consuming the most memory during Edge-Boxes runtime. Moreover, after Edge-Boxes decortication, we observed the possibility that the size of this variable is correlated with the amount of segments clustered during the first Edge-Boxes phase. Recall that such segments are generated due to contours that appear to be disconnected after the edge extraction, and thus they can be related (as a contour as a whole) throughout the affinities variable only. Ideally, those disconnected contours should not appear, and thus we can say that the edge extractor quality is somehow linked to the amount of segments that are clustered. Nevertheless, staying that the lesser the amount of segments, the better the edge extractor, would be also false. For instance, when using an arbitrarily high threshold (or factor) for the magnitude, no edge or segment would be detected, but neither will be any object in the image. Another important point is that the affinities variable grows during runtime, as the number of segments is unknown previously to the clustering. Then, this is why it was difficult to have a better estimation (not based in a worst case) in our previous analysis in chapter 4.

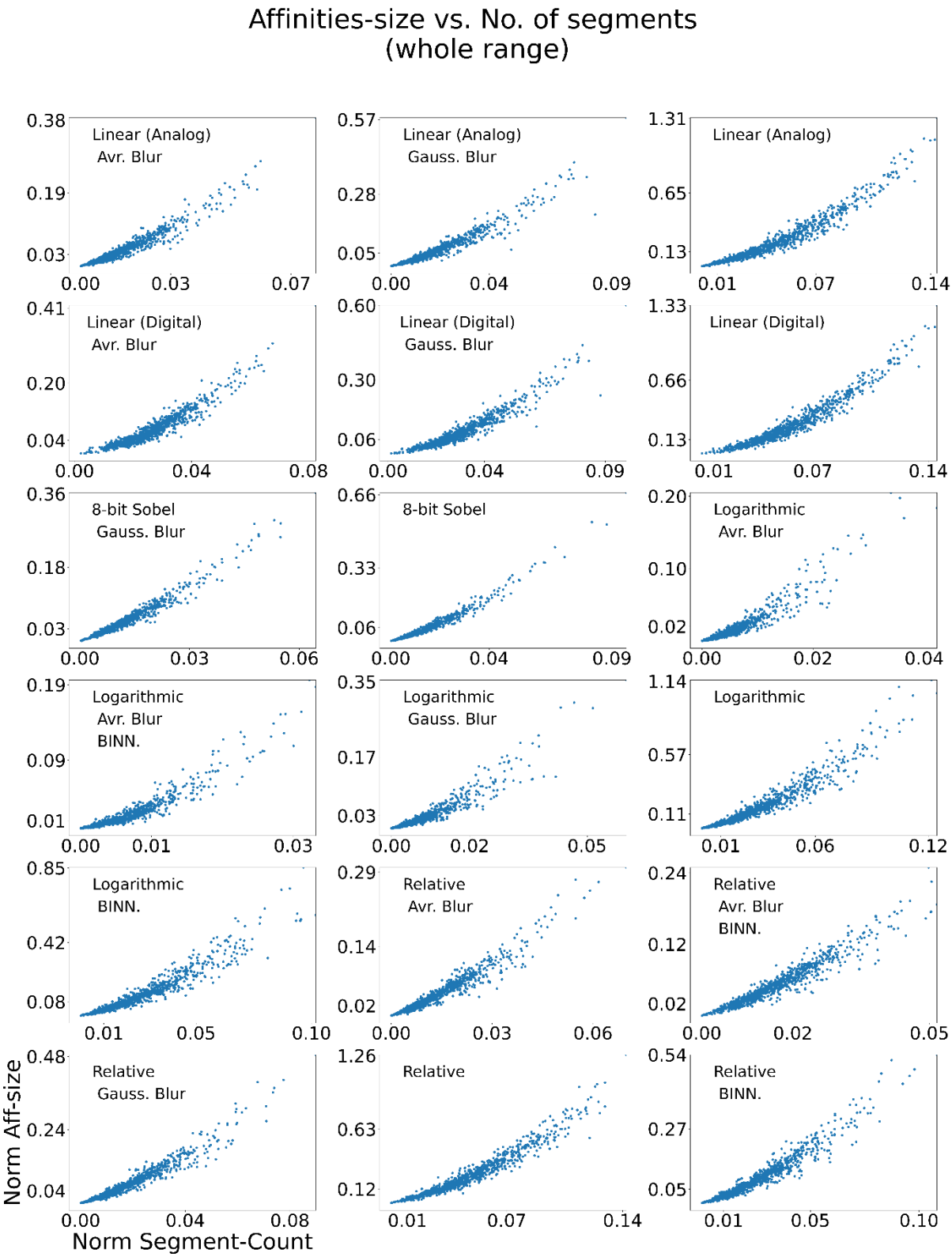


Figure 60 : normalized affinities variable size as a function of the normalized number of egde-clusters (segments) for different architectures. All plots show a positive correlation between the two variables.

In Figure 60, we observe that there is a clear positive correlation between the normalized affinities-variable-size and the normalized-number-of-segments. Remember that we normalized to the number of pixels to eliminate the dependency upon pixels amount.

Blur vs. no blur impact on memory (segment affinities variable size)

In Figure 61, we plot the average of all normalized affinities sizes along each batch of 100 images, and then we average again along the 10 jobs. For the considered architectures, we clearly observe a positive impact of the average blur in reducing the size of the affinities variable.

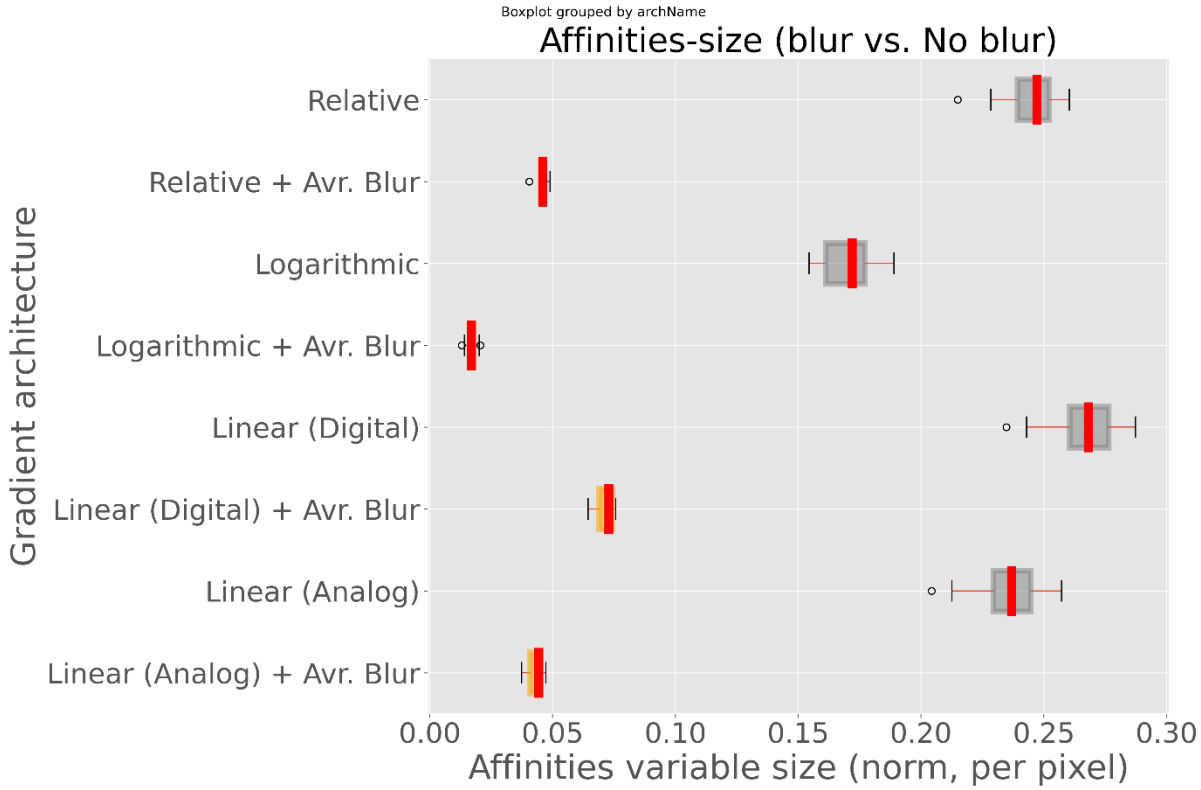


Figure 61 : affinities-variable-size comparison for architectures using and not using de-noising.

Binning vs. no binning impact on memory (segment affinities variable size)

In Figure 62, we observed that, with respect to edge-extractors without an average blur, binning significantly reduced the segments affinities variable size. Moreover, the combination of average blur and binning did not improve significantly (or even increased memory slightly) compared to only binning. Binning alone gives a significant boost for memory, and the averge blur improved it as well. Nevertheless, denoising and binning at the same time gives slightly poorer results when compared with denoising alone. Recall, however, that improving (reducing) memory by means of binning/de-noising could be at the expense of decreasing the ABIoU (performance in localization).

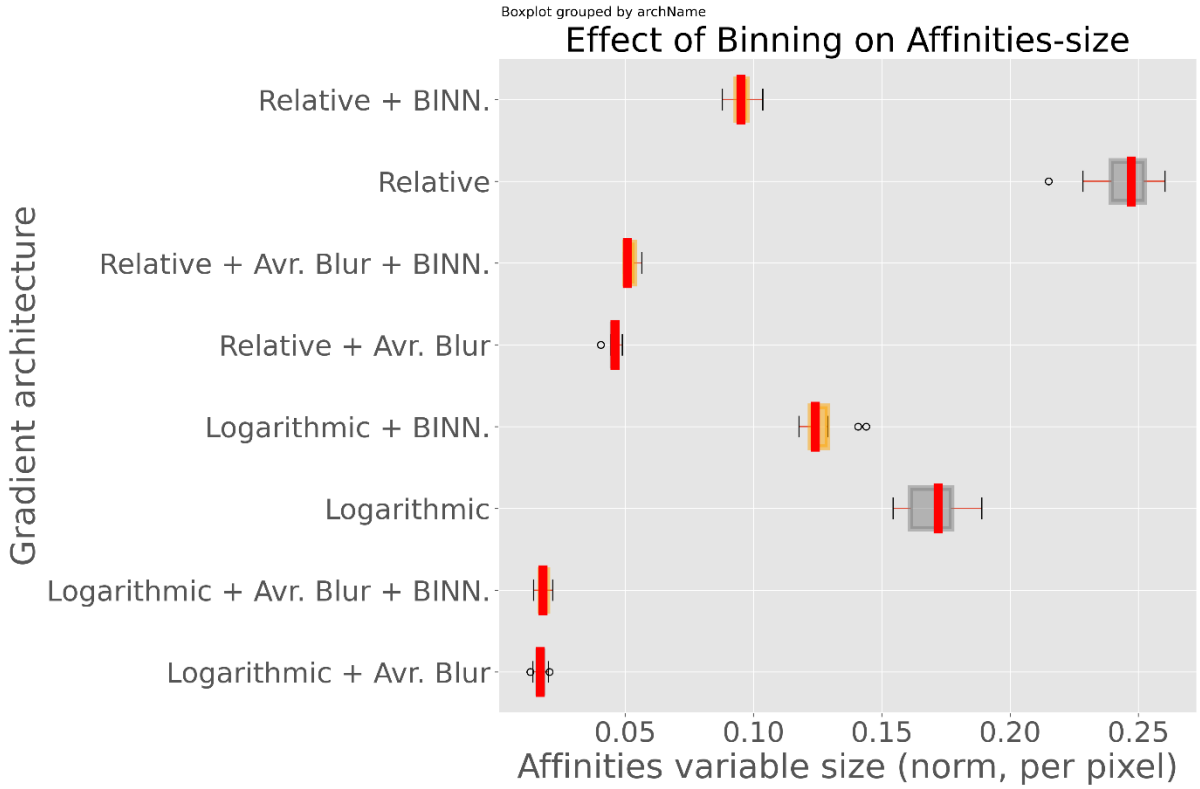


Figure 62 : affinities-variable-size comparison for architectures using and not using 2x2 binning.

Relative vs 4-bit linear gradients

We observe that the memory performance for the architectures in Figure 63 is rather similar for those that did not use a blur step, and for those that use a specific type of blur. For this case, the only point to take into account is that for significant memory reduction either binning of a blur step is needed.

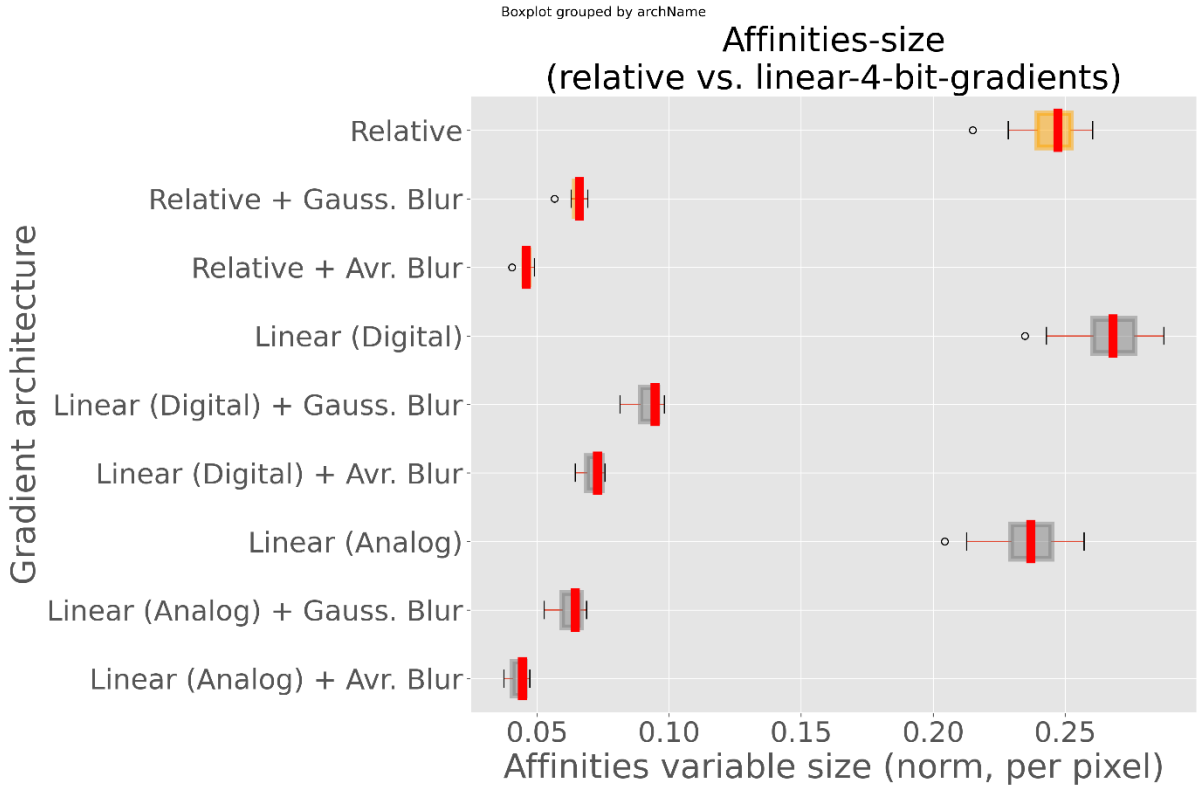


Figure 63 : affinities-variable-size comparison for architectures using relative and simple linear gradients (4 ENOB).

Relative vs 8-bit linear gradients

As in cases before, Figure 64 shows that memory performance for Edge-Boxes execution is similar for ENOBS before edge map generation of 4 and 8 bits, if an average blur is applied before gradients extraction. In this benchmark, the best performance was the relative gradients with an average blur, even surpassing the 8-bit-ENOB case.

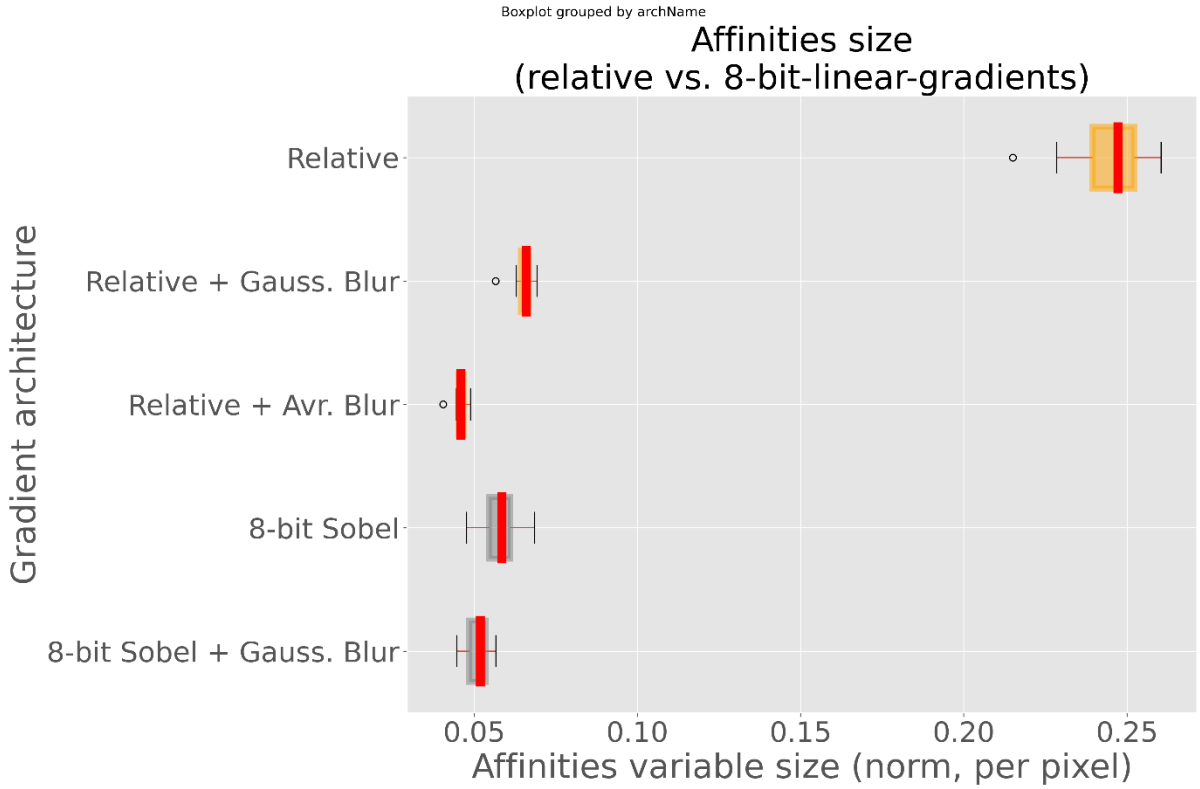


Figure 64 : affinities-variable-size comparison for architectures using relative linear gradients (8 ENOB).

So far in this subsection we have shown mostly qualitative tendencies regarding dynamic memory requirements, and with through an estimation of the affinities-variable size. Now, we go into a more qualitative analysis, and for which we recall the reader an unsolved problem mentioned in chapter 4: indeed, we tried to estimate the total memory required for running Edge-Boxes. Nevertheless, some variables grow in runtime since the number of segments is not known before the program execution. Then, for a preliminary estimation, we hypothesized a worst-case number of segments, equal to the maximum number that can be represented by a non-signed integer data type of 16 bits. Nevertheless, we obtained the average number of segments along all images and jobs (normalized to the image-size) for each architecture and threshold combination. This number corresponds to a coefficient that, once multiplied by the image-size, provides an overall estimation of the number of segments detected by Edge-Boxes. Once we have this number of segments, we can modify it in our estimation from chapter 4. As an example, we present in table # the coefficients we found for the relative oriented gradients (with average blur and with binning):

Table 14 : norm-number-of segments, norm-segAffSize, and global-ABIoU for the Relative Origrad with avr. de-noising and binning.

Threshold	No. of segments (norm)	segAffSize (norm)	Global-ABIou
0.3	0.01620484263724415	0.05186999892981794	0.6948367046975842
0.4	0.012804822681744824	0.03586742498866486	0.6904189331054339
0.5	0.010263935545649961	0.02554154369488491	0.6854328357236822
0.6	0.008351933262945569	0.018695970491177476	0.6737017146185899
0.7	0.006862945858983502	0.013993912559256309	0.6672997980522251
0.8	0.005691222468060574	0.010638890556151607	0.6550939843676324
0.9	0.004759960442535616	0.008248585213333473	0.6434149945568532
1.0	0.004029531099946791	0.006536805260365127	0.6285615565304473
1.2	0.002953513479478474	0.0042213371827614764	0.6030951228860276

In Table 14 we present coefficient allowing estimating the overall number of segments and affinities-variable-size after multiplying them by the image size. For instance, for a 500 x 500 resolution, which we took into account for memory approximation in chapter 4. The overall expected amount of segments is $0{,}01620484263724415 * 500 * 500 \approx 4051{,}51$ segments. Notice that for the worst-case we used segments as $2^{16} = 65536$, which is $\frac{65536}{4051{,}51} \approx 16$ times bigger than the empiric/experimental average value. Moreover, when repeating the exercise from chapter 4, where we approximated the optimal behavior of the C++ compiler, and in order to approximate the memory usage. We obtained an experimental/empiric peak memory of **~5,2 Mbytes** (or ~5,52 Mbytes if we just add the memory requirements for all variables, without trying to simulate the compiler), instead of the ~50 Mbytes related to the worst-case. We also took into account that the input size memory drops to 3 bits times the image size, since we use relative oriented gradients in this example.

The conclusion from this section, is that Edge-Boxes could run on an embedded device having around 5,2 Mbytes of available memory for ROIs generation. Moreover, the pipeline of using Edge-Boxes with strongly quantized oriented gradients is compatible with several architectures, including our relative-oriented-gradients architecture (presented in chapter 5), as well as other in the state of the art (such as logarithmic gradients from (Young et al. 2019). We have presented the impact of two key pre-processing stages, namely binning and de-noising. In general, either binning or de-noising improve the memory usage for our relative-oriented-gradients, where the best performance comes with both combined.

6.2.3 Runtime estimation

In addition, aside from the memory dependence upon the number of segments, we observed that there might be also a link between the runtime (per image and per pixel), the number of segments, and with the affinities size variable. Indeed the more the number of segments, the longer will be the time for scoring each box. As a result, plots for the runtime estimations have the same tendencies as for the affinities variables size. We show the respective benchmark plots below:

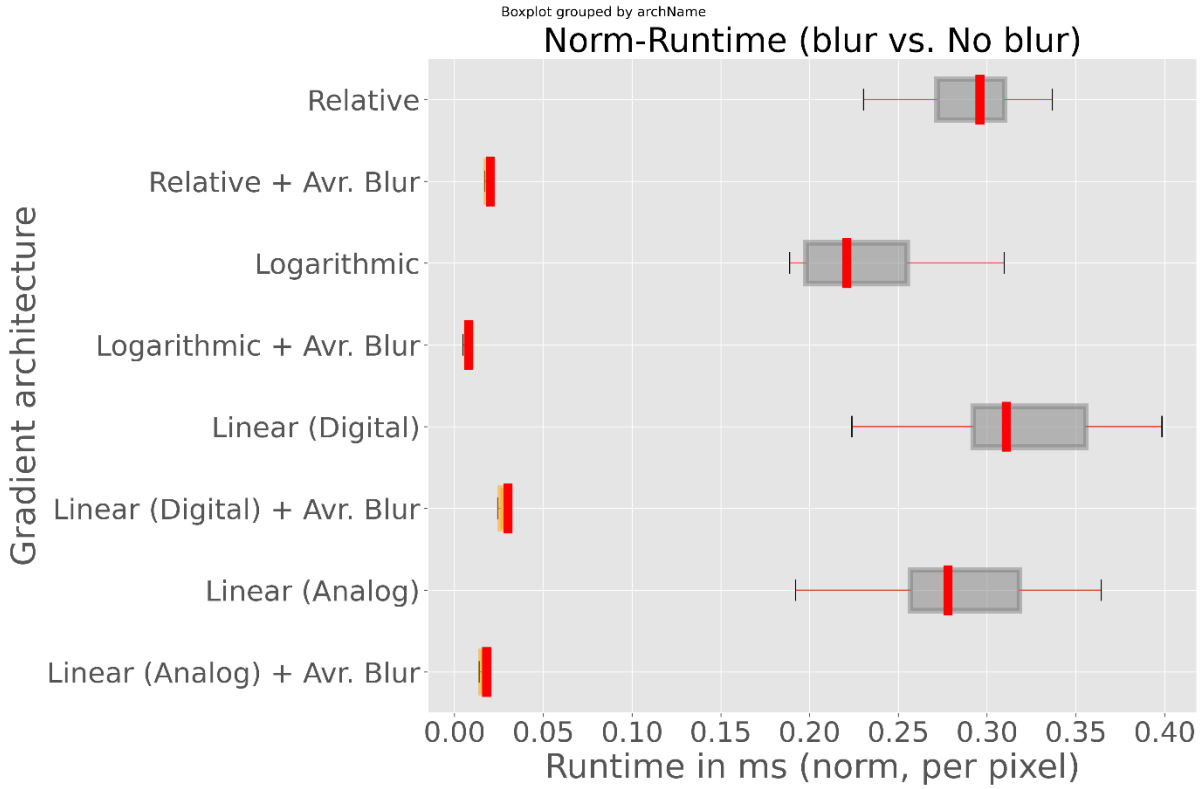


Figure 65 : normalized-runtime comparison for architectures using and not using de-noising.

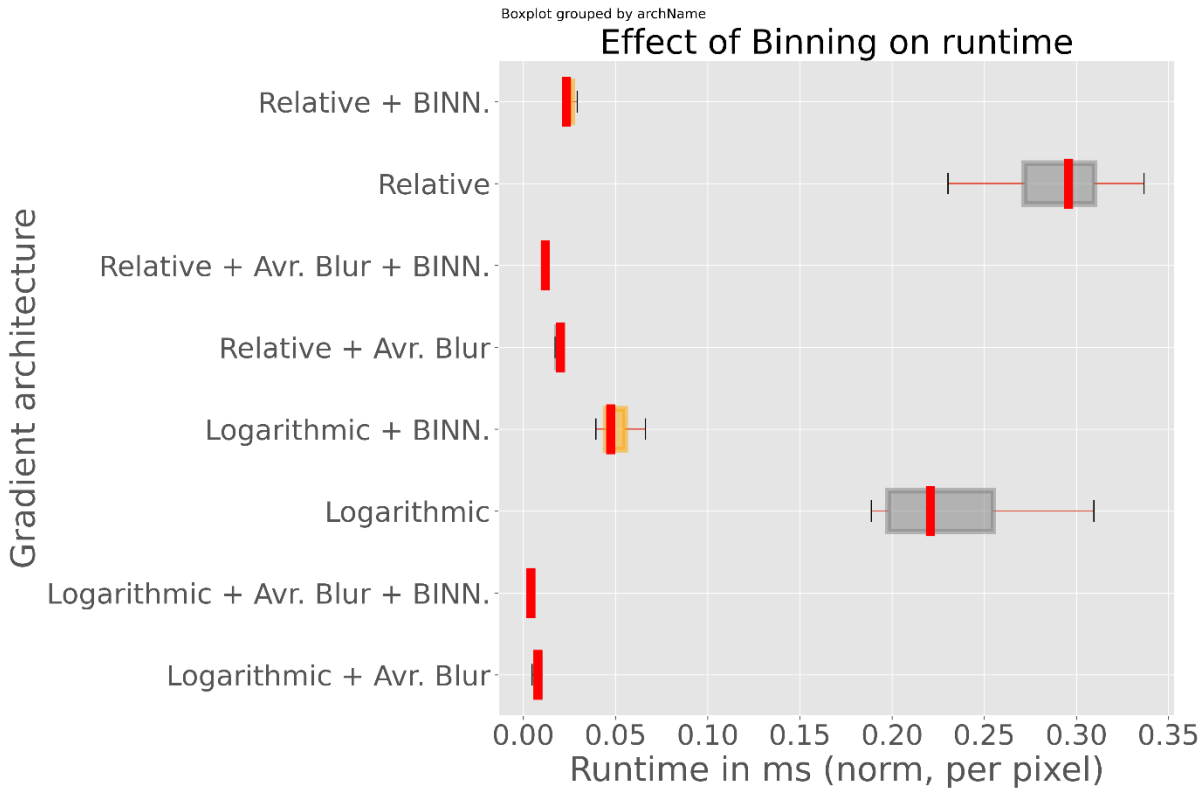


Figure 66 : normalized-runtime comparison for architectures using and not using 2x2 binning.

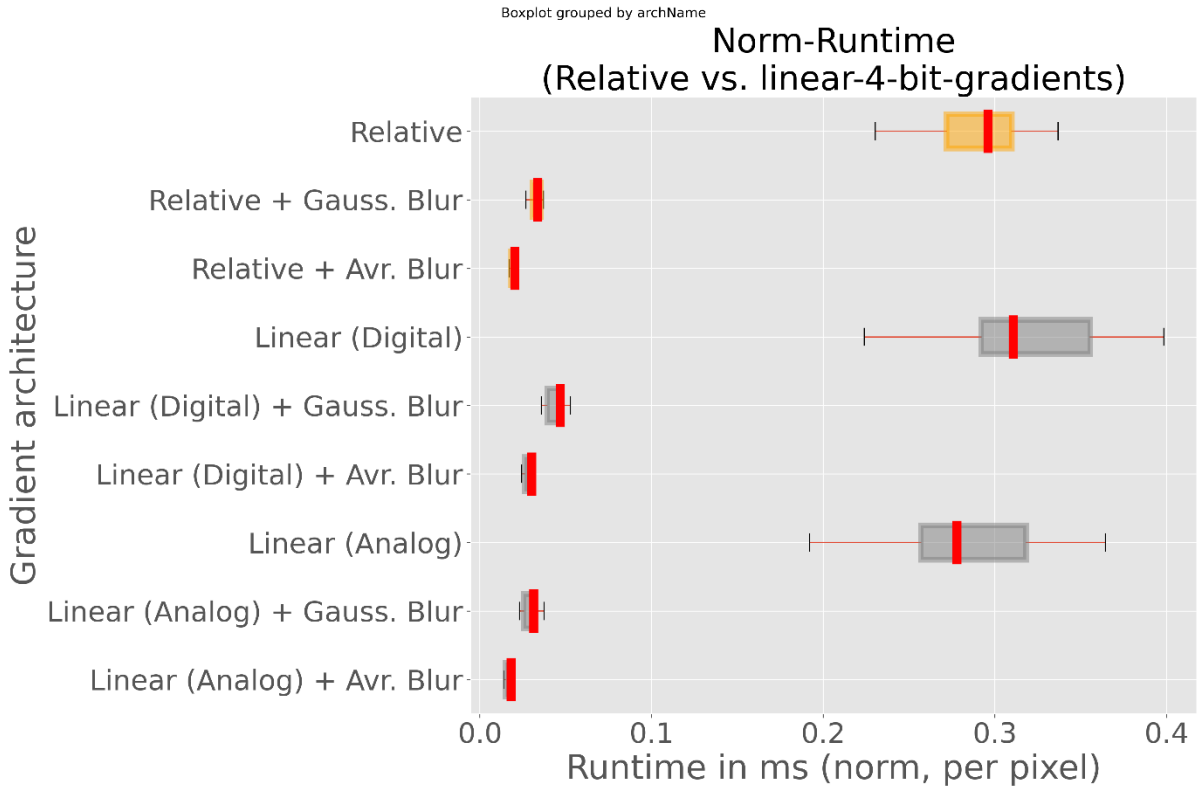


Figure 67 : normalized-runtime comparison for architectures using relative and simple linear gradients (4 ENOB).

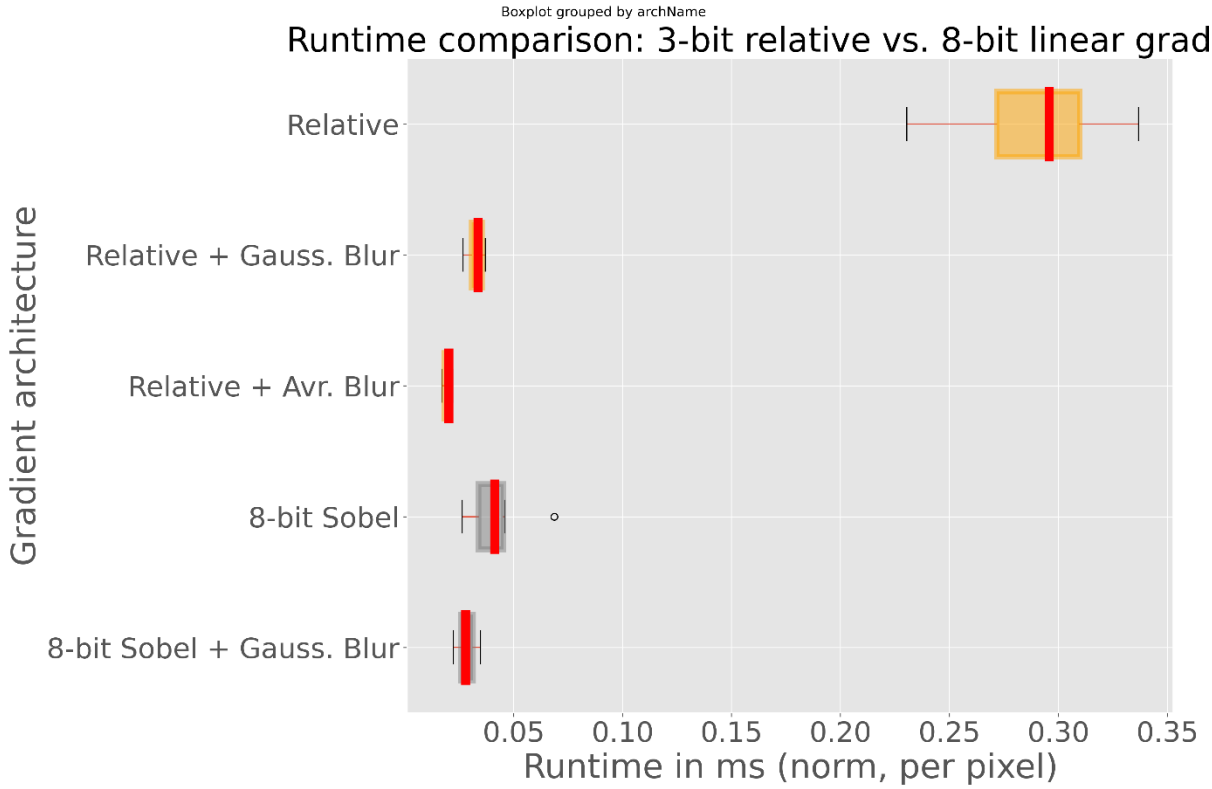


Figure 68 : normalized-runtime comparison for architectures using relative and 8-bit linear gradients.

In this work, we do not try to specifically approximate the runtime of Edge-Boxes, and we only show the difference of running the algorithms with different pre-processing types, and simulated on the same desktop/server machine. Exact runtime values are not so meaningful, and only the relative improvement from one pre-processing to another is useful for design/optimization purposes. The reason is that the runtime depends on the specifics of the system (hardware, and software) into which the algorithm is running. We understand as well that even the shapes/values of plots regarding runtime could vary when changing, for instance, the source-code, the operating system, and the hardware architecture. Thus, runtime-plots are to be considered as only preliminary, and conclusions driven for them are to be taken with caution. That said, we observed that there is a notable positive impact in runtime when using de-noising and/or binning, where the best performance is when both are combined.

6.3. Empiric Figure of Merit

Given that we have three different metrics to characterize (behaviorally) the smart imager system, we then wondered if we could use an empiric figure of merit to jointly characterize each edge extractor type. We thought that the most intuitive way was to take the ratio between the ABIoU and the memory times the runtime (both normalized). However, to improve the range of values, and to give more importance to the ABIoU, we modified it as follows:

$$FOM = \log\left(\frac{ABIoU^{10}}{normRuntime * normSegAffSize}\right)$$

Equation 54

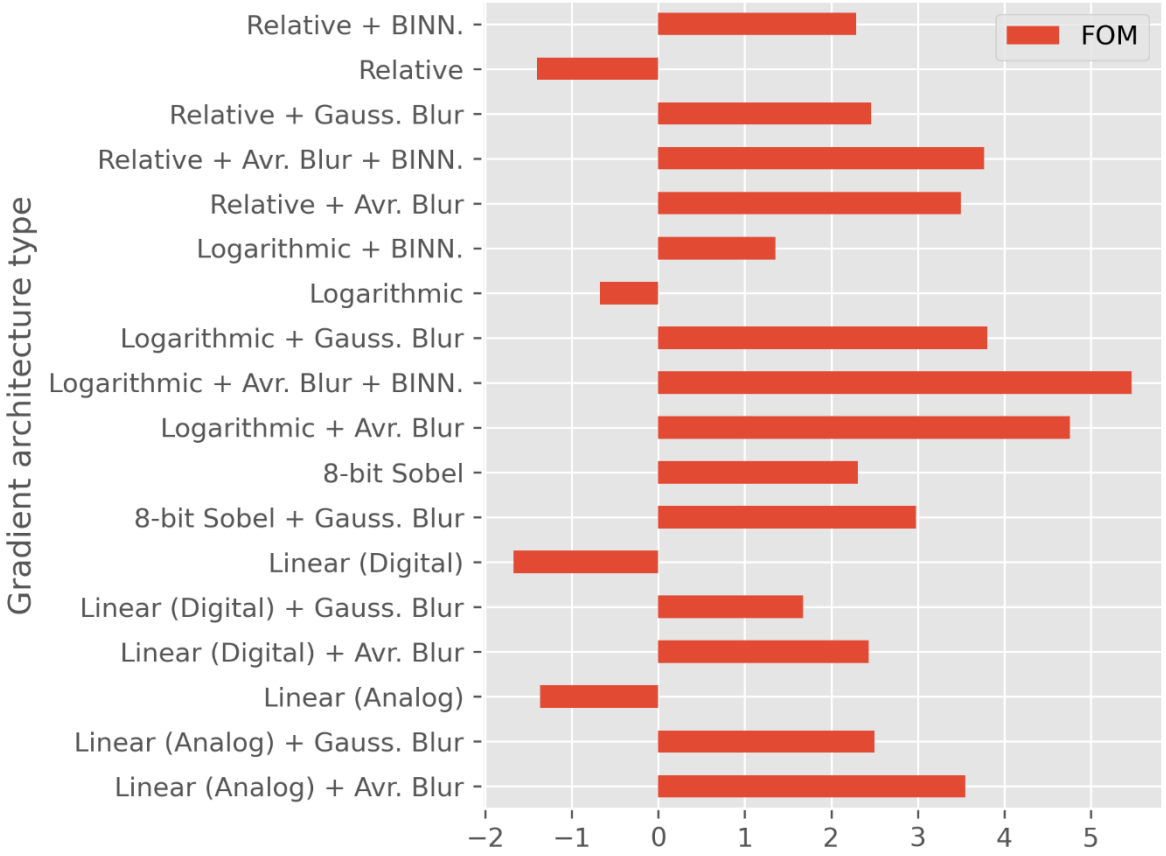
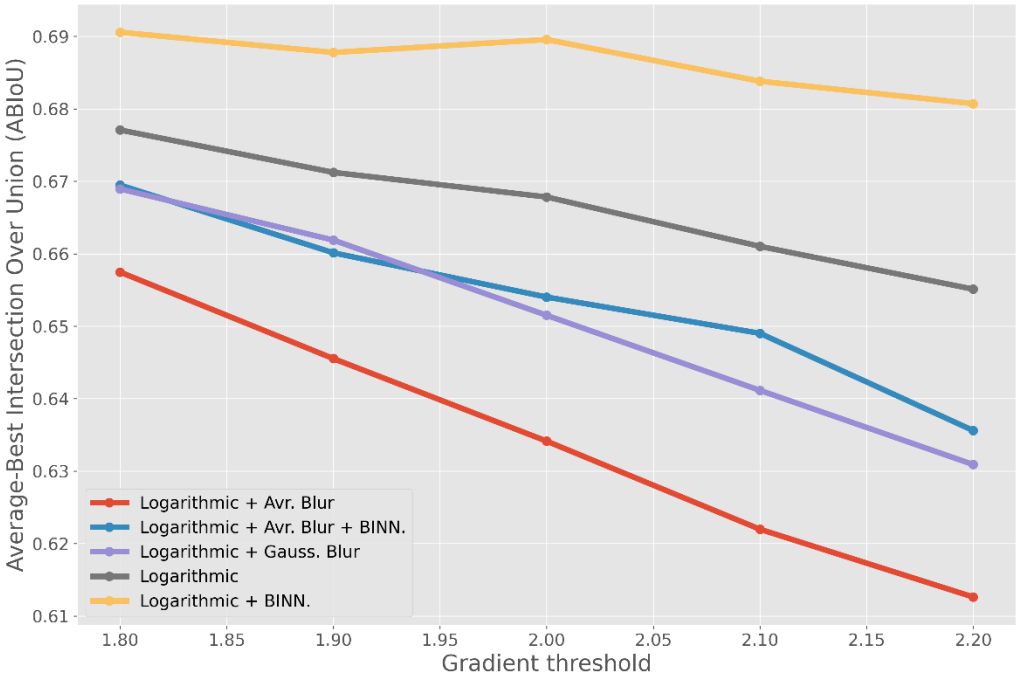


Figure 69 : Figure of merit for different architectures at selected thresholds (from table []).

In Figure 69, we observe that the best FOM was for the logarithmic architecture, with an average blur kernel and with binning. However, this architecture also had one of the worst ABIoU. That is why we observed that even though we used an exponent of 10 for the ABIoU in the FOM, yet the equation makes the impact of the normalized segments affinity size and the normalized runtime more important. This also raised the question that, in the case that this FOM was accepted since memory and runtime are the most important parameters, then one could change the initial strategy for threshold selection before in the chapter, since it was based uniquely on ABIoU performance. That is illustrated in Figure 70.

(a)



(b)

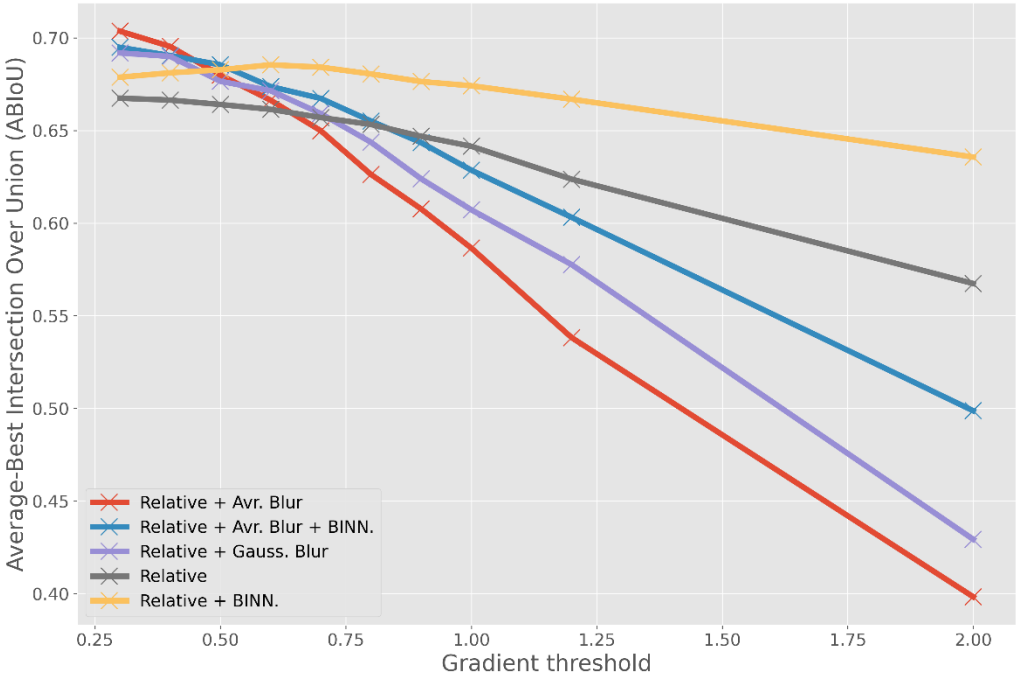


Figure 70 : Global ABIoU dependance upon the gradiend threshold for different architectures.

In Figure 70 (a) and (b) we observe the ABIoU tendency for different relative and log architectures when varying the threshold value (log cases) or the threshold gain (relative cases). As expected, and systematically, the ABIoU reduces when increasing the threshold, since a lower amount

of edges is detected and more disconnected contours appear. Now, we can observe what happens with the FOM when we increase the threshold:

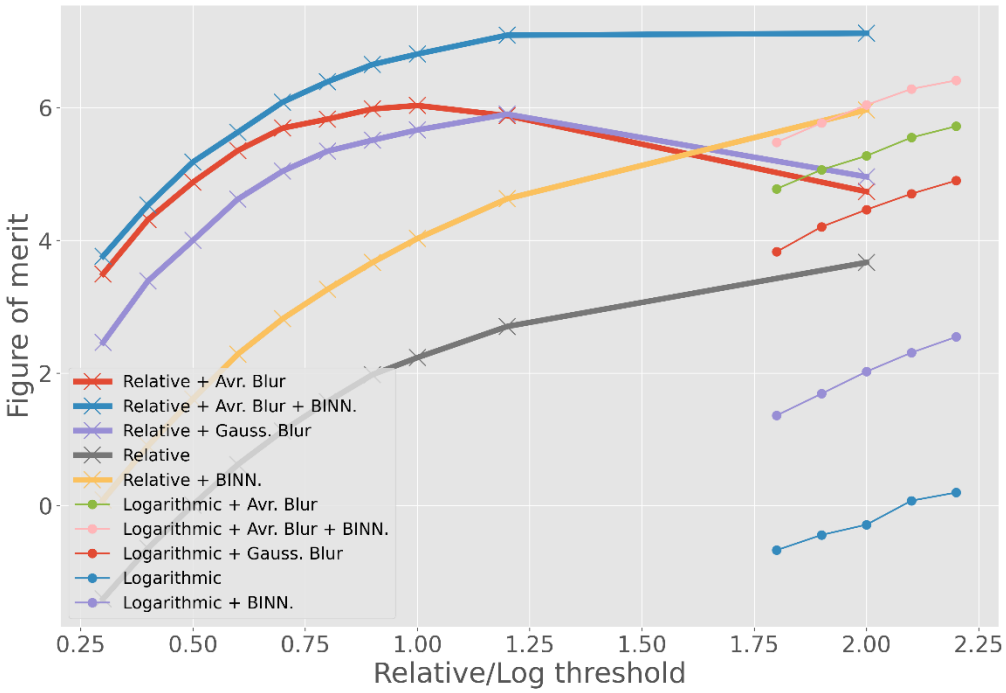


Figure 71 : figure of merit as a function of the relative/log threshold for different architectures.

In Figure 71, we observe that the FOM systematically increases (except cases in which the relative/log threshold is so high that it starts significantly diminishing the performance), reflecting thus the impact of runtime and memory reduction, but does not reflect (clearly enough) the diminishing on the ABIoU. To circumvent this issue, we finally decided to plot the FoM versus the ABIoU:

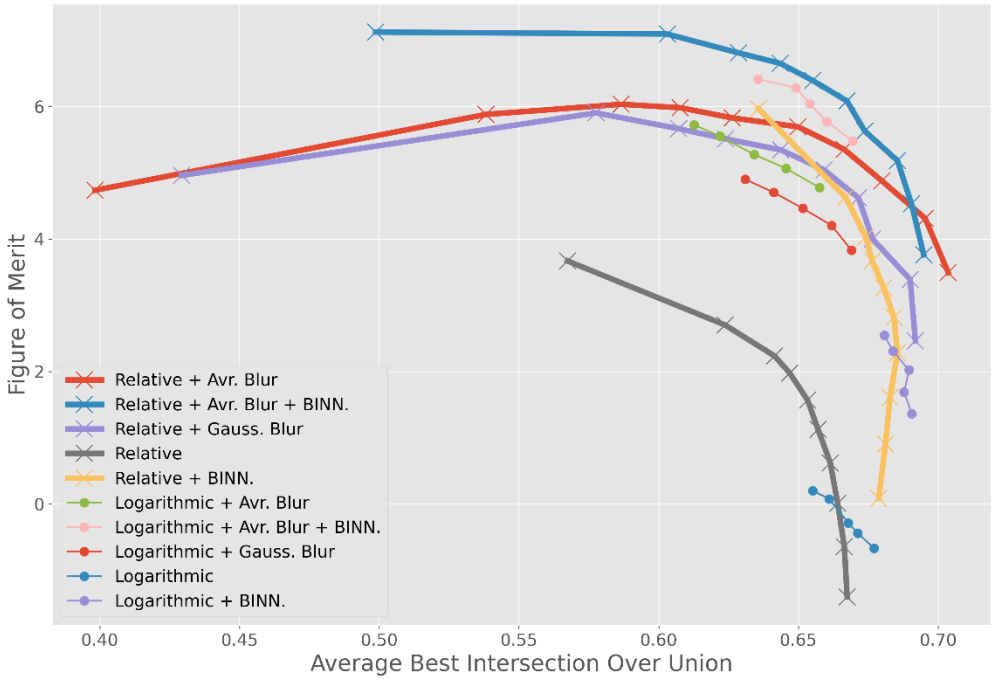


Figure 72 : dependance of the figure of merit upong the ABIoU for different architectures (both quantities are parametric funcitons of the architecture thresholds).

In Figure 72, we plotted the FOM when varying the ABIoU (being the threshold factor the hidden parameter). Then, we can "draw" an imaginary vertical line representing a desired ABIoU, and for this ABIoU we then have the corresponding FOM for each architecture. From this plot we observed that not all architectures (at least in the parameter sweep we did[28]) achieve the maximum ABIoU.

6.4. Conclusions of chapter 6

In this chapter, we presented our scalable simulation scheme. In addition, we provided benchmarks for system performance in terms if IoU, memory usage through affinities-variable-size, and runtime estimation. We proposed a way of choosing each architecture threshold for benchmarking based on the ABIoU. Benchmarks illustrated as box plots allow a first estimation of memory usage and relative time complexity in comparison to other edge-extractors (in correlation with corresponding ABIoUs). Moreover, we proposed a FOM based on those three variables, and we studied it in order to to give it a proper interpretation. We plot the FOM vs. the ABIoU when the architecture threshold is varying. Then, we draw an imaginary vertical line representing a constant (desired) ABIoU. Thus, each

[28] It is possible that, for example, the logarithmic gradients could achieve a higher ABIoU, which, based on the figure of ABIoU vs. threshold factor, would mean decreasing the threshold factor. That, however, could lead to higher memory and runtime needs. Such case scenario could be analyzed if found relevant in further works. Notice as well that for the case of logarithmic gradients, we did a factor sweep around the value 2, recommended by the original authors (Omid-Zohoor et al. 2018; Young et al. 2019).

intersection with the vertical line with each architecture curve gives the FOM. The upmost architecture corresponds to the best architecture, corresponding (for the parameter sweep we made) to the relative variant of our model, with an average blur before edge extraction, and with a 4x4 pixel binning.

Indeed, since the output from the edge-extractor is aggressively quantized, one reasonable question is if Edge-Boxes could also exploit such quantization to reduce the bit-depth during computations. We think that this could be possible, but we let this question for further works.

Chapter 7. Dynamic Vision Pre-processing

In last chapters, we have discussed about classic or frame-based approaches for object detection. However, there is a specific motivation to assess the viability of OD with neuromorphic or dynamic-sensing imagers. Indeed (Gori 2018) *mentions that solutions for OD based on single static frames is far from how nature copes with this challenge. Specifically, living beings seem to benefit from motion information as well. That's the reason why they say that using both static and dynamic information could make OD an easier problem to solve. In this chapter, we will explore such idea.*

7.1. Event-base data for object detection

One possible way to use dynamic sensors for OD is to use their inherent motion detection to obtain set of region proposals like done by (Acharya, Padala, and Basu 2019). Their mechanism aims at spotting regions containing clusters of events potentially containing an object, as shown in figure Figure 73.

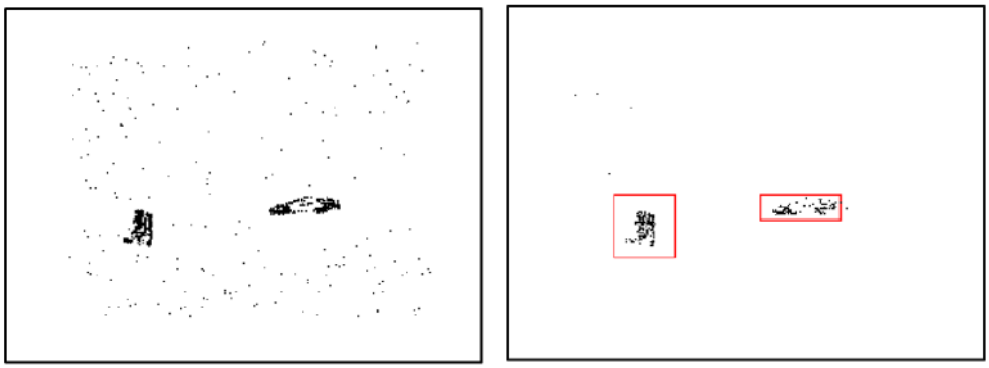


Figure 73 : example of regions proposals generated in (b) from (a) (Acharya, Padala, and Basu 2019).

Moreover, examples like (Deng, Li, and Chen 2020; Deng, Chen, and Li 2021) show that classification tasks are also possible with dynamic data codified as asynchronous spikes. In addition, in they work, they propose generating frames (or more complex data-structures similar to frames) from the asynchronous event-streams, and then use more conventional, frame-based learning approaches. Another way of using machine-learning with events, and which could exploit the asynchronous events-behavior, is by using a family of approaches called Spiking Neural Networks (SNNs). For instance, (Lamba and Lamba 2019) present a comparison between CNNs and SNNs for handwritten-digit classification. However, as mentioned by (Barchid, Mennesson, and Djeraba 2021), SNNs algorithms are not as mature as FB counterparts, and so it is not clear for us how to use SNNs for multi-class and multi-scale object detection, and at the same time exploiting neuromorphic (event-based) imaging-systems. Indeed, (Barchid, Mennesson, and Djeraba 2021) show an example of SNNs performing single-object detection, but their work is, from our perspective, still only a prove of concept which has to be further studied at algorithm level. In addition, (S. Kim et al. 2019) showed that SNNs can indeed accomplish complex tasks beyond classification, such as OD. They (S. Kim et al. 2019) used a way of converting deep neural networks to spiking neural networks and attained good performance for OD. Nevertheless, they used the PASCAL VOC dataset (Mark Everingham et al. 2015), which is frame-based, and then it is not obvious for us how to map this technique to neuromorphic imaging systems. Because of the reasons mentioned in this paragraph, we think that, at the time of the present work, algorithms for OD (with supervised learning techniques) are not mature enough when using neuromorphic (spiking-based or event-based) algorithms. However, neuromorphic image-systems are still interesting in this work since they allow obtaining dynamic data in the image directly, which could allow, for instance, exploiting such dynamic information for attention-based mechanisms for more efficient ROI proposals generation. In next paragraph, we start covering the state of the art for the so-called Dynamic Vision Sensors (DVS).

There are several imager variations trying to recover dynamic information. Such sensors typically correspond to the DVS family as they share several features: firstly, “scene-motion” is

captured by detecting temporal light-intensity variations for every pixel. Secondly, the sensing of those changes happens at pixel level. Thirdly, the way for communicating those changes (namely events) is as spikes, from which the system recovers specific information (pixel address, time stamp, and polarity[29]). Finally, the DVS scheme abandons (although the asynchronous read-out is not mandatory, as implemented by (Li et al. 2019)) the routine of sequential-synchronous-photo-integration plus read-out, for a spike-asynchronous-read-out. This asynchronous read-out of incoming events, codified as spikes, is managed by a dedicated digital-circuitry called the arbiter.

Regarding last paragraph, one interesting question is why the DVS scheme could be more interesting that just subtracting image frames. Firstly, the event-based approach, where only pixels detecting events send a spike, allows saving data-throughput from regions where nothing is happening. Secondly, the combination of in-pixel event-detection, plus asynchronous reading allows outstanding low event-detection latencies (in the order to tens of us (Lichtsteiner, Posch, and Delbruck 2008; Berner et al. 2013)). Finally, DVS pixels have been shown to achieve temporal contrast sensitivities for instance between 1% (Yang, Liu, and Delbruck 2015) and 0.3 % (Delbruck and Berner 2010) . The temporal contrast (TC) of a dynamic vision sensor is the threshold at which the pixel "detects a change" and sends a spike (Posch et al. 2014). Such change is typically described as the relation between the light intensity changing respect to the last spiking time, and the light intensity changing also at the last spiking time (Posch et al. 2014). However, for us, it is not clear if a more sensitive TC is essential for OD. Nevertheless, achieving the aforementioned features by subtracting frames (in an embedded and frame based manner) implies somehow buffering images, and subtracting them at an elevated frame-rate for a later comparison with a threshold (where this threshold can vary for each pixel alone depending on the intensity at the local last event). For instance, considering the 15 µs spike latency reported by (Lichtsteiner, Posch, and Delbruck 2008), the equivalent frame rate for achieving a similar latency would be of $(15\ \mu s)^{-1} \sim 66.7\ kfps$. In addition, subtracting images one after the other is not enough since events can be related to changes slower than the frame rate (an event at time T_i can depend on light intensity at time T_{i-2}, if we imagine a fine-grained discretization of the continuous time-line, with time steps[30] $\Delta T = T_i - T_{i-1}$). Thus, there has to be added logic for handling pixel-values-updating in the previous buffered-image. Finally, the subtraction should happen at a SNR or bit-resolution (depending on the implementation being analog or digital) sufficient for the required contrast sensitivity or temporal contrast TC. Those constraints do not make obvious for us the implementation of a low power imaging device with a classic frame-based approach.

In this chapter, we try to assess if a DVS sensor could *improve* the OD pipeline we have studied so far. We preferred to have an imager that does not rely specifically on SNNs, since they are still young research topic, whereas Convolutional Neural Networks or Support Vector Machines are extensively proven to work for frame-based OD. Moreover, another solution could be collecting events during a period and then performing classic, FB based OD on this output, but then this seems more like a FB type of processing, and it is not clear for us why to use a DVS in such case. In addition, the in-pixel read-out significantly degrades the fill factor, while the asynchronous read-out imposes a complicated read-out scheme. Thirdly, the event data-throughput, measured in events per second (e/s), can easily increase not only due to motion along the scene, but also because of photonic-noise, illumination changes, shadows and texture-artifacts, making the power saving or the OD not that efficient. In fourth place, the stream of events, if coming from different occluded objects, has to be somehow segmented

[29] The polarity is a Boolean value indicating whether the light intensity derivative is positive of negative.
[30] That is, indeed, the strategy we adopted for simulating this kind of neuromorphic imaging-systems.

for each particular object (if not, then it would be more difficult to know how each object is moving). Finally, the last question is how to perform OD if the scene is (sometimes) static.

Our strategy for coping with the complications mentioned above was the following: we proposed a simulation scheme with synthetic datasets. From our simulations, we expect to see if we can improve one specific problem: the high event data throughput, because, from our point of view, it was critic for reducing redundancies and the corresponding power consumption. Then, we try to create a Frame Based imager that efficiently "emulates" part of the behavior of the DVS, so the emulated dynamic vision part will make the FB one more efficient and fast regarding the object localization task.

7.2. Synthetic dataset generation

As mentioned in chapter 3, our preferred strategy for behavioral DVS simulations was using synthetic datasets at a high frame rate (~ 1 kfps or more). The reason why we had to create our datasets is that we wanted to evaluate the impact of changing the system at pre-processing level (pixel level, or at bottom-column). Moreover, we wanted to evaluate specific points, all at the same time, that were hard to attain in datasets in the state of the art. For instance, at the same time that we wanted to simulate changes at pixel or column-bottom level, we also wanted to evaluate the behavior of different edge extractors (and Edge-Boxes) when objects are of different size, when shadows are present, when lighting conditions around the image differ significantly, and when there are both moving and static objects. Also, we wanted to start from rather simplistic datasets which could allow to drive simpler and preliminary conclusions, instead of using real-life video-datasets from which, potentially, observations would be harder to interpret. Then, for approaching as much as possible to the real behavior, we preferred using a simulator taking into account light phenomena on a custom scene. For such purpose, we hypothesize that each frame represents the instantaneous photocurrent generated at each pixel. In this section, we focus on how we generated the two datasets we used in this work.

7.2.1. Polygon dataset

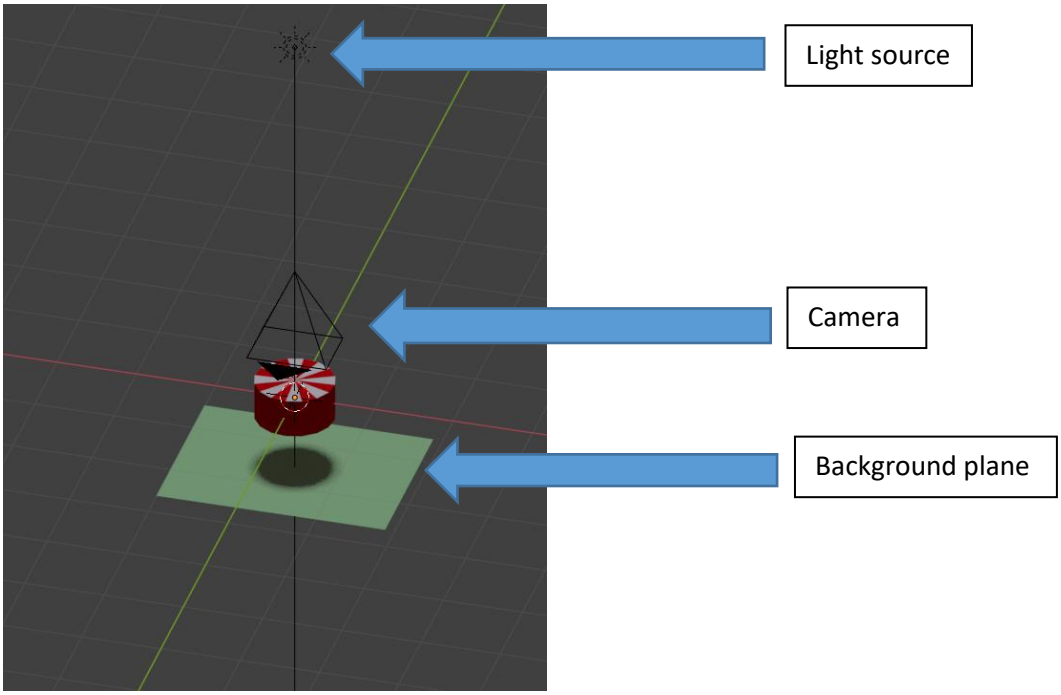


Figure 74 : our animation setup for generating our first synthetic dataset.

The simulation setup in Blender is shown in Figure 74. It corresponds to a cylinder that is rotating around its main axis, a camera capturing only the top view, and a light source. As mentioned in our previous publication (Cubero et al. 2020), we rendered 1000 frames for a 37-degree rotation of the cylinder along the Z axis. In Blender, we configured the amount of frames and the motion, but the frame rate can be chosen arbitrarily. For our case, the fps can be derived by considering a hypothetic event read latency (which dictates the time resolution of the simulation, and that is why we relate it with the frame rate for simulation purposes). In our paper (Cubero et al. 2020), we used $15\ \mu s$, inspired by (Lichtsteiner, Posch, and Delbruck 2008). Then, the fps is the inverse of the time resolution, giving approximately 66,7 kilo-frames per second, which gives to our cylinder a rotating speed equivalent to that of a car-wheel, with 0,5 m diameter, going forward at ~77,5 km/h. In addition, for rendering the video sequence, we used the Cycles renderer, and contrast threshold of 3 %, which was an arbitrary yet illustrative value chosen to be in the range of reported values from (Lichtsteiner, Posch, and Delbruck 2008) and (Berner et al. 2013). One example of a rendered frame from the setup presented in Figure 74 is presented in the Figure 75.

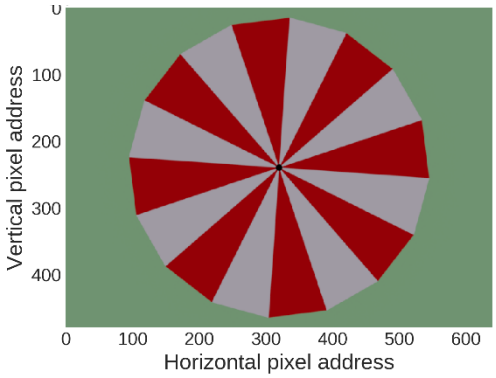


Figure 75 : example of a rendered image from the video sequence generated with the simulation setup from Figure 74 (Cubero et al. 2020).

One interesting experiment was to observe the resulting light-intensity curve evolution in time for one particular pixel. In the figure below, that is what we present for two particular pixel positions: one pixel was closer to the center and it corresponded to a smoother (lower gradient) curve. The second one was closer to the polygon perimeter, and it corresponded to a higher gradient curve. Both pixels where in the same radial line and their values were taken during the same period of time.

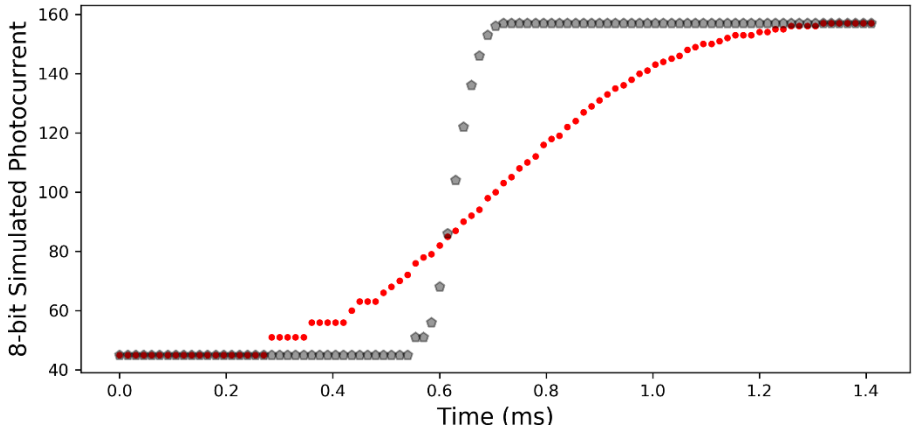


Figure 76 : light intensity evolution for two pixels when a color-edge is passing through. The gray plot corresponds to a pixel far away from the polygon center, whereas the red one corresponds to a pixel closer to the center (Cubero et al. 2020).

From Figure 76, we observe that we arrive to generate a light-intensity curve at high frame rate. This kind of curve is the one that will be used to approximate the events for each pixel. Notice also that the pixel that is closer to the perimeter corresponds not only to a higher slope in the intensity versus time curve, but it also has a higher linear speed[31]. Then, last experiment agrees with the hypothesis that (at constant and approximately uniform illumination conditions) the curve slope is correlated to the particle-speed in the image[32].

[31] As a reminder, the linear speed (magnitude of velocity) of a particle describing a circular motion in 2 dimensions is the scalar multiplication of the radial distance times the angular speed.

[32] Saying that the slope is correlated to the speed in the 3D World is incorrect, since that would not take into account projection effects. Then, we cannot obtain directly the object speed in the real World with such simplistic approach.

7.2.2. Three spheres over ground dataset

Our synthetic dataset consisted in three spheres of the same size located in front of the camera. Each one was at a different depth distance from the camera, so their size in the image plane would change respect to the others. In addition, we rendered a video-sequence, in which each sphere showed a different type of motion. One sphere moved faster than the others, another one was the slow sphere, and another laid static. We also located a background plane perpendicular to the image plane, so the shadows would be visible. From this dataset, we wanted to be able to simulate it with both edge-extractors, but also with a DVS architecture. Moreover, we made the scene (see Figure 77) such that objects would appear at different sizes because of the distance from the camera, they would move differently (or not move at all), and there would be different lighting conditions along the rendered images (see Figure 78). As mentioned before, all those requirements made hard for us to find a datasets that would comply with them at the same time.

This dataset was generated similarly to the case of the polygon dataset. Nevertheless, this dataset was for testing the object localization task, besides of behavioral simulation of DVS imagers.

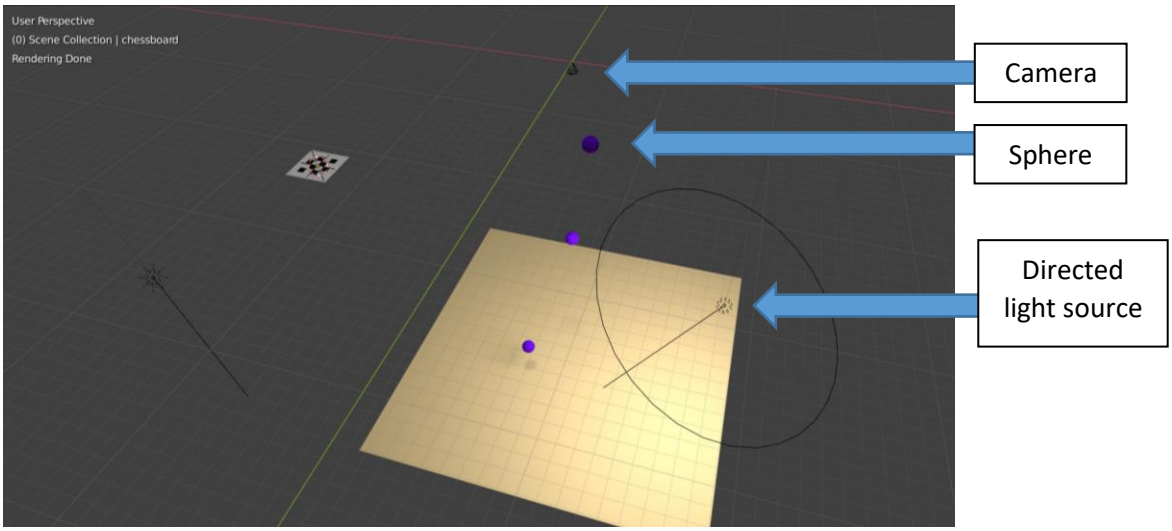
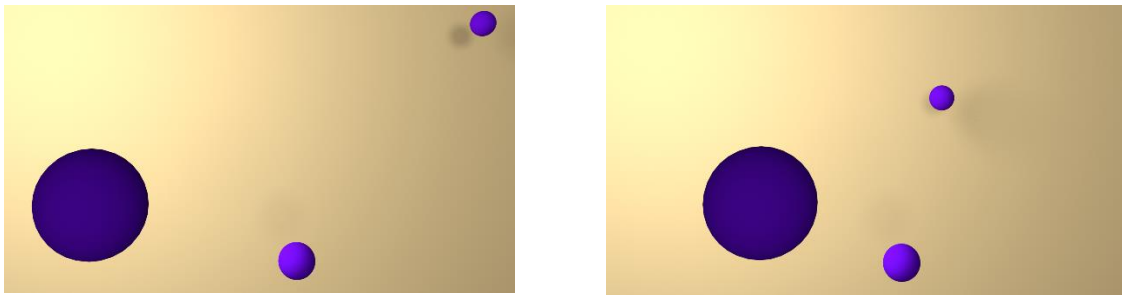


Figure 77 : our animation-setup for our second dataset generated.



(a) Frame 0 (b) Frame 59

Figure 78 : first (a) and last (b) frames rendered from the video sequence related to the setup in Figure 74.

Since the video sequence corresponds to a list of frames, extracting edges for each frame can also be carried out (in order to simulate Origrad architectures), which we illustrate in Figure 79.

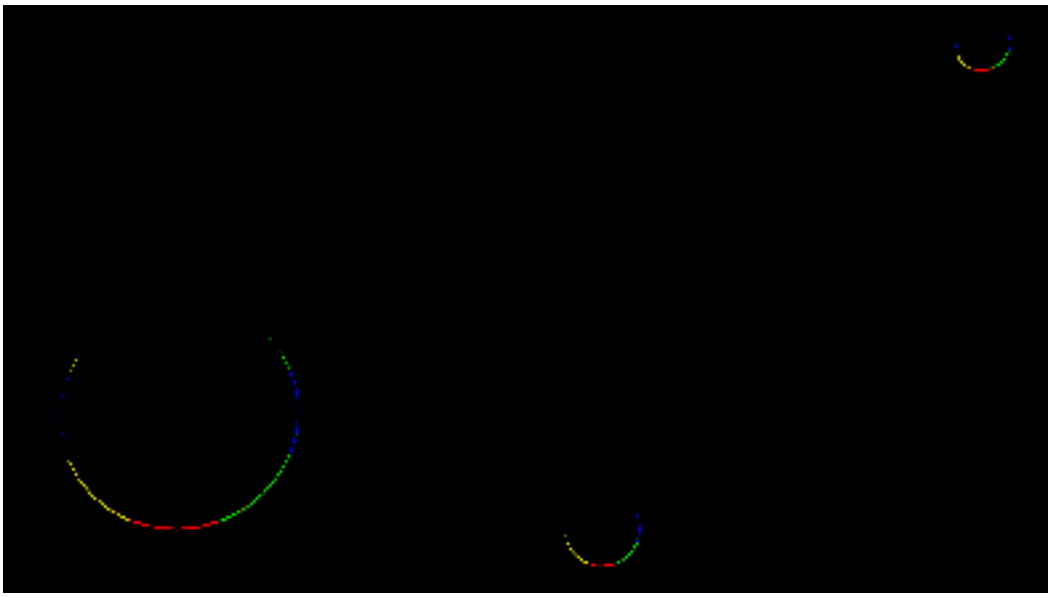


Figure 79 : example of a rendered frame from the output of the relative-edges extractor. Colors indicate the detected edge direction (blue: 90 deg, red: 0 deg, yellow: 45 deg, green: 135 deg).

Nevertheless, for object localization benchmarking, we need ground truth bounding boxes as well. Specifically, for each frame, we need to know the coordinates for the enclosing rectangle for each different object (sphere) in the image. However, in Blender, we specify object-locations in 3D World coordinates (X, Y, Z)[33], and the size (radius) in simulation units for each object. Moreover, we define the camera position, a parameter related to the focal distance and the image (size) resolution. Notice also that Blender uses the pinhole camera model. Then, the problem we faced was to automatize the ground-truth bounding-box generation during the frames rendering, so we could compare the output from the object-localization algorithm (Edge-Boxes) with the ground-truth. The way we did it is described in the next paragraph.

Before we start solving the problem, we summarize it as follows: given the 3D World position and orientation coordinates of any object of any form and size, and given the camera position and orientation, we need to obtain the coordinates of the ground-truth bounding-box in the image plane. This is a problem of projective geometry, and we start from the classic camera calibration method described in ("Camera Calibration" 2021), which use the known formula for the pinhole camera model (OpenCV-dev-team 2012):

$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 1 & c_x \\ 1 & f_y & c_y \\ 1 & 1 & 1 \end{bmatrix} [r|T] \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Equation 55 : pinhole-camera equation (OpenCV-dev-team 2012).

Last equation states the basic form of the pinhole camera model. In order to explain better how we used it, we will make reference to the illustration in Figure 80. There are two reference frames involved: one for the camera, represented by {C}, and one for the World represented by {W}. Those

[33] Aside the 3 rotation angles, which in this case are not relevant since our simulation objects are spheres.

two are not necessarily aligned. Now, considering a point p in Blender, we define the vector $\vec{P}_{\{W\}}$ of point p in World coordinates. We start with a point before we go into shapes that are more complicated. In addition, we define the position vector $\vec{C}_{\{W\}}$ (not represented in Figure 80) of the camera origin respect to the World. There, we understand that there is another reference frame whose origin is at $\vec{C}_{\{W\}}$ and that is aligned with the camera. Then, the image plane is at one focal distance f from the camera. Since we are dealing with a projective geometry problem, we have to express $\vec{P}_{\{W\}}$ in homogeneous coordinates, and then we have to transform from reference frame {W} to {C} by multiplying $\vec{P}_{\{W\}}$ by a rotation-translation matrix. That is :

$$\vec{P}_{\{C\}} = [r|T]\vec{P}_{\{W\}}$$

Equation 56

$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 1 & c_x \\ 1 & f_y & c_y \\ 1 & 1 & 1 \end{bmatrix} \vec{P}_{\{C\}}$$

Equation 57

We observe that the pinhole-model equation represents the projection of the point in camera-coordinates in the image plane. Coordinates u, v represented the x, y position in the image. The factor s comes from the homogeneous coordinate's normalization factor, so the third entry equals to 1. The matrix relating $\vec{P}_{\{C\}}$ and the left-side of the equation is the camera-matrix, which contains the intrinsic camera coefficients: f_x and f_y are the focal distances for their corresponding axis, and (c_x, c_y) correspond to the image center. In our case, $f_x = f_y = f$. Moreover, in order to simplify the problem, we can make {W} and {C} to correspond, thus making the derivation of $[r|T]$ straightforward. Notice that this alignment is practical because we are working on a 3D-scene simulator. However, in realistic applications there may be scenarios that are more complicated.

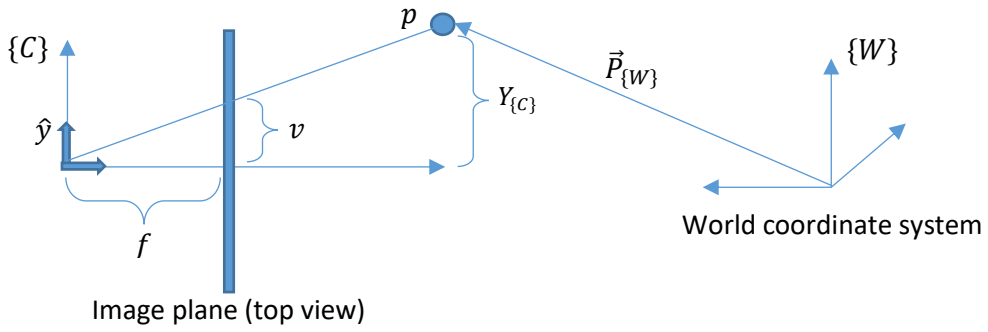


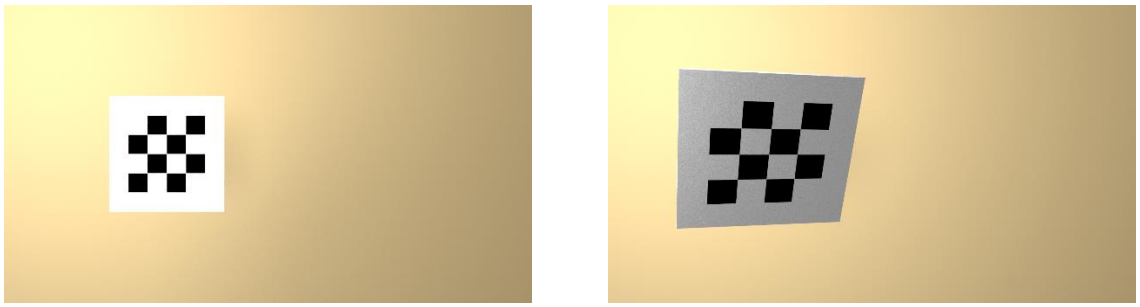
Figure 80 : example of how the image plane is placed into the World coordinate system.

We have advanced in solving the problem stated before, yet there are questions unresolved: firstly, we have stated the equation for a point, but we have not yet generalized for more complicated 3D-shapes. Secondly, how do we obtain the intrinsic camera parameters f, c_x, c_y?

We start by addressing the first issue by considering that Blender codifies objects of any complicated shape as collections of 3D points with different types of relations amongst them (3D-

meshes). Different relations can lead to the formation of lines, planes, and volumes. Thus, for avoiding a more complex geometric problem, what we did was to make a Python script that would output and save, for each object (sphere) the position of each vertex for each object[34]. Then, we expressed the object coordinates in camera coordinates, and finally used Equation 55 to get all projected points in the image. Finally, generating the ground-truth bonding box for each object was straightforward, since all we had to do is to take the maximum and minimum image point coordinates for each collection of vertices associated with each object.

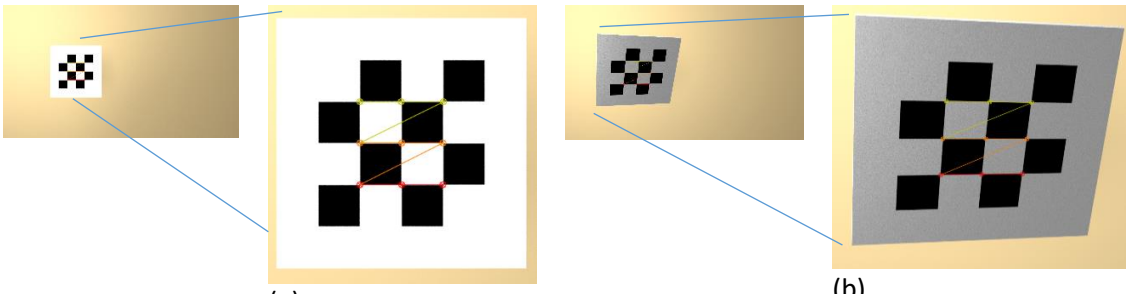
The second question relates to a typical camera calibration problem. Indeed, we were not sure if we could obtain the physical value of the focal distance from Blender directly. Then, in order to be sure of the correct value, we proceed to a camera calibration process as explained in the OpenCV documentation for real case scenarios ("Camera Calibration" 2021). The process consists of using a grid of points whose relative position is fixed (relative to a reference frame attached to the grid origin). One typical example is a "chessboard" pattern, where the particular points are the intersection between squares. We modeled such pattern and animated it so we would have the chessboard at different positions respect to the camera. This process is illustrated in Figure 81.



(a) Frame 10 (b) Frame 20

Figure 81 : example of two rendered frames when animating a moving chess-board pattern in the scene.

Then, by keeping the camera fixed and moving the grid (so the particular points appear in the camera in a projected manner), one can use one of OpenCV's functions for recovering the intrinsic camera parameters, if for each recorded frame the position of such points in the image plane is known. However, we still had to know the position of each particular corner point in the image plane. In order to simplify the problem, for each rendered image containing the chessboard, we used an OpenCV function for finding the chessboard points in the image (as showed in Figure 82). Then, we used those points (coming from the same grid, moved along different positions in different sequence frames) as input for the camera calibration function.



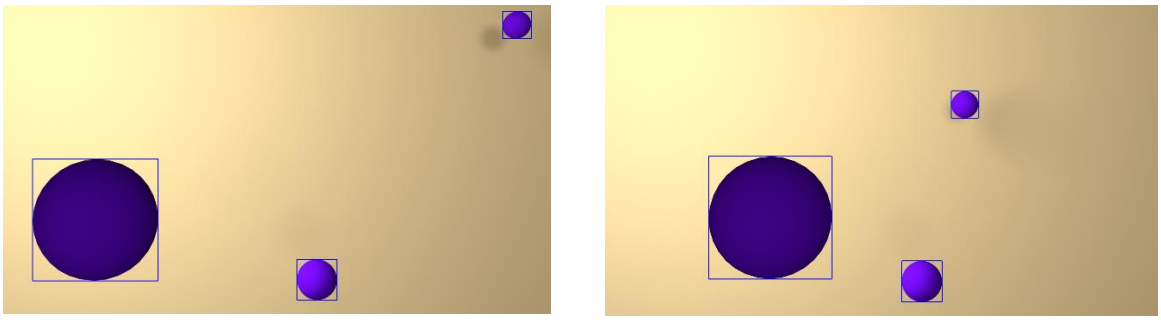
(a) (b)

Figure 82 : result from OpenCV function for detecting the chessboard pattern corners at frame 10 (a) and at frame 20 (b).

[34] In order to do so, we had to « move » the objects in the scene for animation programmatically as well.

That last one gave approximated values for the focal distance and image center. Since this calibration process is not perfect, we had to refine the values (by trial and error) to match satisfactorily human-made ground-truth boxes.

After the process made in last paragraphs, we successfully obtained a synthetic dataset for object localization with ground-truth annotations generated automatically, as illustrated in Figure 83. Further works can exploit such strategy for modeling more complex and general problems, since Blender is currently capable of using photorealistic scenarios, including (for example) humans, animals and vehicles.



(a) Frame 0 (b) Frame 59

Figure 83 : example of the first (a) and last (b) rendered frame from the animation-setup from Figure 77, and with ground-truth bounding-boxes generated with the method previously described.

In next sections, we explain how we used those synthetic datasets in order to simulate the DVS and several variations.

7.3. From circuit schematic to behavioral simulations

In order to make a fair behavioral model of the DVS, we studied first the basic circuit schematic for a single pixel. Notice that there are several DVS variations with different specifications and pixel complexities. We took, however, as reference the circuit proposed by (Lichtsteiner, Posch, and Delbruck 2008), and we present it in Figure 84:

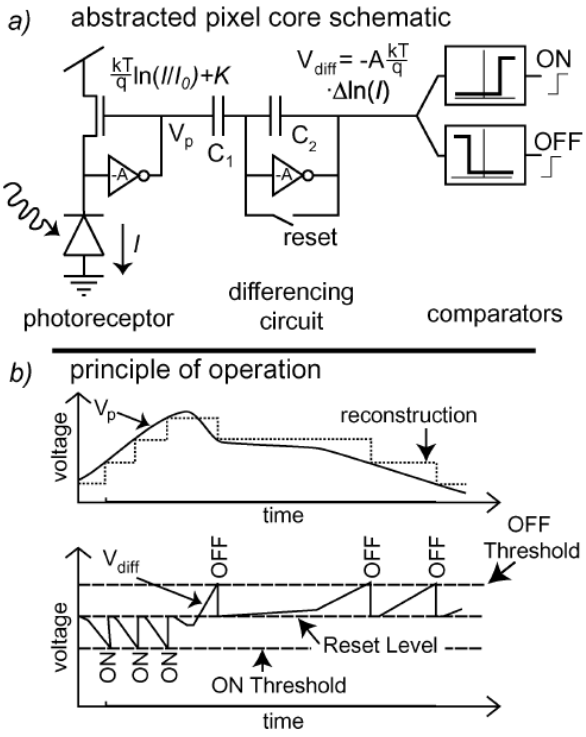


Figure 84 : "(a) Abstracted pixel schematic. (b) Principle of operation. In (a), the inverters are symbols for single-ended inverting amplifiers." (Lichtsteiner, Posch, and Delbruck 2008)

Part (a) Figure 84 shows the basic pixel schematic from (Lichtsteiner, Posch, and Delbruck 2008), while Figure 84 (b) present its principle of operation: each time the relative temporal differences reaches the temporal contrast, a spike in sent and the reference value is reset. In Figure 84(a), the schematic is partitioned in three steps: photoreceptor stage, difference stage, and event-detection-stage by means of a threshold comparison of the (relative) lighting difference. When the reset signal is closed, the voltage across capacitor C_1 follows the logarithmic output V_p. Once the reset is open, the capacitor C_2 starts cumulating the charge associated with the logarithm of the current intensity minus the logarithm of the intensity at the time the reset switch was open. Such signal is represented as V_{diff} in part (a) of the last figure. That is:

$$V_{diff} = -A\frac{kT}{q}\left(\ln(I) - \ln(I_r)\right)$$

Equation 58 (Lichtsteiner, Posch, and Delbruck 2008)

Where I_r corresponds to the intensity at last time the reset switch was opened (t_r), and A is a factor reflecting the equivalent amplification showed in Figure 84 (a). Then,

$$V_{diff} = Const \cdot \ln\left(\frac{I}{I_r}\right) \approx Const \cdot \left(\frac{I - I_r}{I_r}\right)$$

Equation 59 (Posch et al. 2014)

So V_{diff} approximates the ratio of the relative lighting difference respect to the intensity I_r at time t_r. The final stage compares the differentiating voltage with two thresholds. Those two thresholds relate (in absolute value) to values representing the so-called "temporal contrast" (Lichtsteiner, Posch, and Delbruck 2008). When $|V_{diff}|$ is higher than the threshold, a "ON" request signal (e.g. a spike) is sent if $V_{diff} > 0$, and a "OFF" spike is sent otherwise. Moreover, when a spike is sent, the arbiter then

"acknowledges" the spike reception, and triggers the reset switch to close temporarily, so the reference lighting intensity updates. In the simulation, this process has to be followed for every pixel independently, since each pixel may follow a different intensity curve, have a different lighting-intensity-reference, and spike at different time stamps (this is why simulating a DVS by only subtracting subsequent frames and comparing such difference with a positive, or negative, threshold is incorrect). Figure 85 illustrates the output from a DVS simulation with a temporal contrast threshold of 3 % (when events where cumulated during 15 µs.

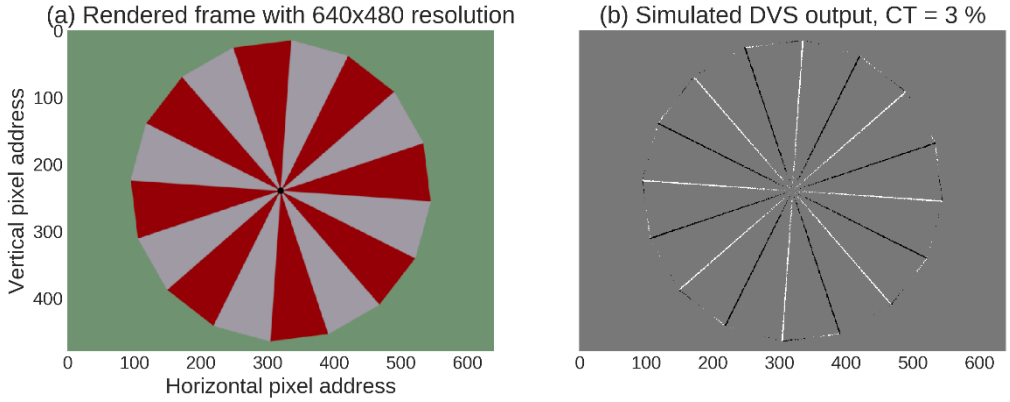


Figure 85 : (a) example of a rendered frame from our first (video-sequence) synthetic dataset, and (b) example of a rendered frame after behavioral simulations of a DVS (Cubero et al. 2020).

7.4. Pre-processing improvement for data-throughput reduction

Notice (if we want to make a correspondence with frame-based approaches) that we cumulate the stream of spikes during a short period, and render them in an image such as the one above (b). Then, the rendered image appears to follow the shape contours. From that perspective, rendering frames with spiking data and obtaining an edge-map from a static image could potentially be similar for object classification. The difference is that the edge-map contains also the edge orientations, which are not directly present in the spike-rendered-frame. On the other hand, with extra processing more dynamic information could be obtained, for example velocity directions for different particles in the scene. Nevertheless, that approach would require (to our knowledge) using temporal correlations, which goes further from the frame based approach that we have been using so far before this chapter.

Regarding the localization task, one intuitive approach would be to cluster events from a rendered frame from spikes. The clustering criteria could be simply the Euclidean, or Manhattan, distance between pixel addresses, and we think (since we did not test it) that a simple-enough algorithms may do the task satisfactorily for this dataset. Nevertheless, this dataset is too simplistic for saying that basic clustering approaches would work in real datasets, which may present occlusion, complicated textures, lighting irregularities, shadows, and complex object motions. Such aspects may appear in the spike-stream as events whose nature is not directly known, and more post-processing (spatial or temporal) may be required in order to obtain meaningful information. Then, for us, it was not clear if the spikes processing required for dealing with specific issues related to a spike-stream output would actually be advantageous when comparing it to a standard frame-based approach.

Indeed, we can formalize what was introduced in last two paragraphs as two problems that we address in this work: firstly, we wondered if we could reduce the amount of events or events data

throughput. However, at the same time we wanted to reduce the information loss of neglecting some events. Secondly, we wanted to be able to use both spatial and temporal processing at the same time. However, we observe that the frame based approach for Origrad differs significantly in pre-processing and architecture from the DVS approach. Moreover, Origrad gives directly meaningful information (e.g. edge magnitude and orientation) for localization and classification, whereas frames recovered from a DVS output may require potentially complex pre-processing before localization and classification could be performed. Nevertheless, the DVS output potentially allows attention based mechanisms for ROI proposals, and fast detection of moving objects. We tackle the first problem by introducing the modulation we proposed in our previous publication (Cubero et al. 2020), and represented in Figure 86:

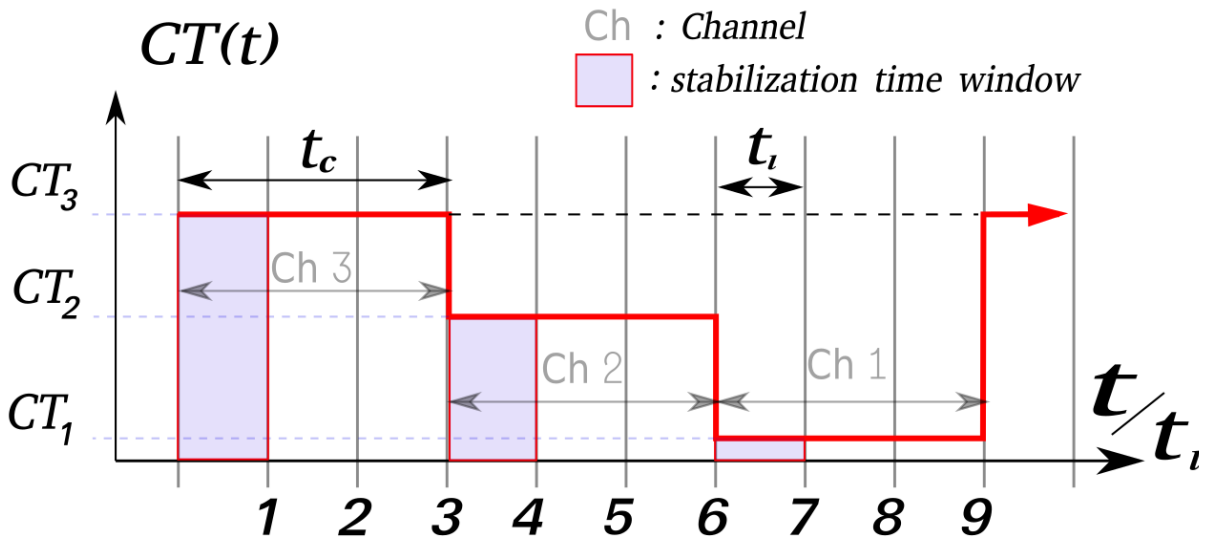


Figure 86 : our proposed modulation scheme (Cubero et al. 2020).

In Figure 86, we present our modulation scheme (Cubero et al. 2020). CT corresponds to the global contrast threshold, named "temporal contrast" (TC) for instance by (Lichtsteiner, Posch, and Delbruck 2008), which corresponds to a voltage reference used in the comparators in Figure 84(a). The reason for varying CT along time is inspired in the curve from Figure 76. Being t_l the simulation time resolution (15 µs), numbers from 1 to 9 represent different time stamps at which the simulation updates. Those updates track new events, so they approximate an asynchronous behavior when the scene is evolving much slower than t_l. The modulation consists in a stepwise lowering of CT along several (in our case, 3) values. The amount of time a CT remains constant is t_c or the "channel time". For prove of concept, in our experiments we set $t_c = 3 \cdot t_l$. For practicality, we call each set of events corresponding to each $CT_{i \in \{1,2,3\}}$ a "channel". Then, 3 channels relate to 3 different CT values along a complete modulation period. Indeed, a particle at the image plane that moves fast generates a higher intensity slope when passing by a particular pixel (in the circumference described by such particle motion). In practice, this higher slope could lead to more events detected per unit of time, and we may make the hypothesis that several events coming from the same intensity slope (e.g. the same pixel) are not adding relevant information, only redundant information. Of course, this hypothesis may only hold to a certain extent, since eventually the smart imaging system may "check" the state of motion or intensity evolution at a place of a high slope. One possible optimization is to just to "disable" spikes from pixels detecting events at a high rate (e.g. observing a high intensity slope). Nevertheless, that

introduces the problem of how to measure such frequencies per pixel without significantly changing the original pixel schematic. We did not take such approach, and instead we tried to lower the probability of high-slope pixels generating events by setting CT to a high value, namely CT_3 in Figure 86. This action has two consequences, pixels with higher slopes will tend to spike less, and pixels with lower slopes will be much less likely to generate a spike. Then, channel 3 corresponds to pixels with high slopes, yet its throughput is statistically less due to the relatively high global CT. However, lower-slope pixels may be also relevant, and they start appearing in other channels by stepwise lowering CT. Nevertheless, by decreasing CT, higher-slope pixels start increasing in frequency again. Even though there is already a data throughput reduction thanks to the implementation of a high CT during a certain time, the problem of data-throughput increasing after decreasing CT can be improved, as we explain later in this section. For now, we will discuss about a caveat of abruptly changing CT as in Figure 86.

The need of the stabilization window

The reader may have notice that in Figure 86 we presented specific time steps in light-blue, which we indicated as "stabilization-time-windows". In fact, during our experiments, we observed a relatively high peak of events throughput arising each time that CT was abruptly changed. Our explanation for such behavior is based on Figure 87:

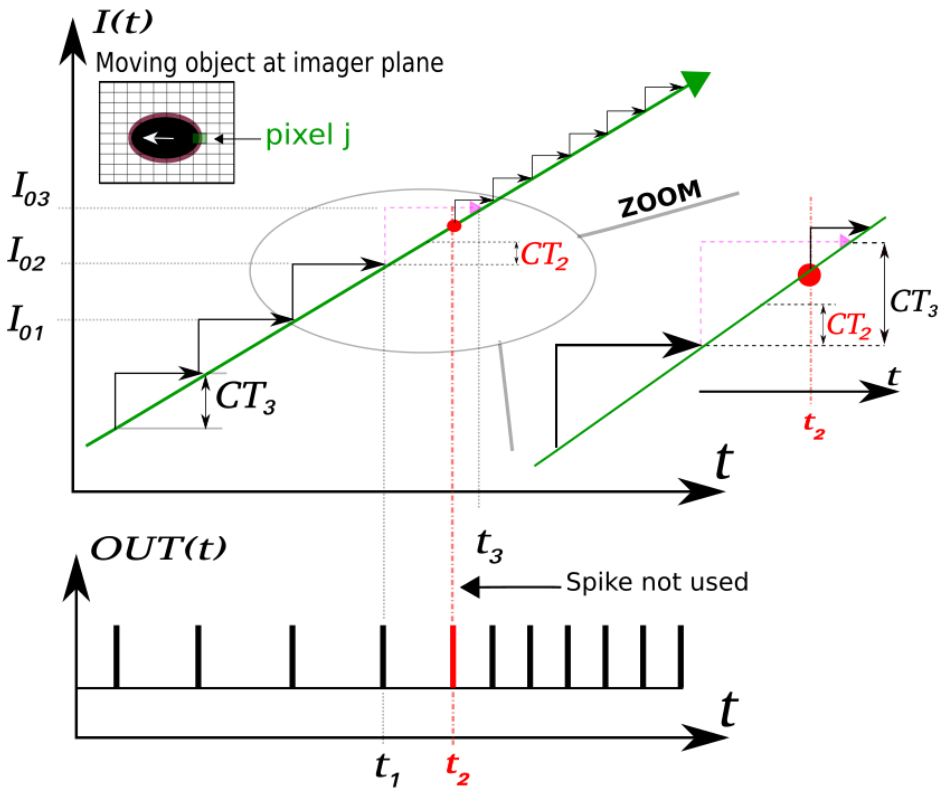


Figure 87 : illustration of how an abrupt changing in CT triggers undesired spikes at the moment of the changing (Cubero et al. 2020).

Figure 87 presents a simplified version of an intensity curve as a straight line. The horizontal axis corresponds to time, and we include (below the plot) a representation of when spikes are

generated: for each moment that the curve attains a relative change equal to CT_i, an event is detected and a spike is sent. Moreover, for each event, the local reference value for the light intensity updates to the value at the last event. Notice that, as CT_3 is relatively high, it "prevents" lower-slope pixels to detect events. However, their V_{diff} values could be increasing (decreasing) as well, but not enough to trigger a spike. Then, when CT decreases abruptly from CT_3 to CT_2, several pixels that were "in their way" to attain the threshold reach it simply because the global CT became suddenly significantly lower. One example is depicted by the red spike in Figure 87. Such spike might not be irrelevant, but its interpretation is less straight forward than in the case of spikes not happening due to sudden changing in CT. For keeping our solution simple, we propose to just discard events coming from the time stabilization window. Then, we do not take into account such events for benchmarking different imagers throughput.

Further optimization with events inhibition

As we mentioned before, when CT decreases, higher slope pixels increase their spiking frequency since they become more likely to attain the threshold. Nevertheless, they may have already generated at least one spike during the previous channel (with a higher CT). Then, if the hypothesis that the real scene is evolving much slower than t_l (so that all intensity-curve-accelerations during a whole modulation period are approximately zero[35]), then it is reasonable to inhibit spikes coming from pixels that have already spiked in a previous channel. We observed two different ways of implementing this idea: firstly, any pixel that spikes during a channel is inhibited from spiking until next channel. Secondly, any pixel that spikes during a whole modulation period is inhibited until the next period. More specifically, the way the inhibition works is by changing the pixel schematic so it independently becomes "unable" to spike after generating a spike. This inhibition can only be removed by a global (synchronous) signal, which we call the synchronous enable (SE). Thus, pixels can get inhibited asynchronously and independently, but they are all reset to the abled state together and at the same time. In next subsection, we explain how to change the pixel schematic in order to implement both the modulation and the inhibition.

Pixel schematic changes

In Figure 88, we present both the original pixel schematic from (Lichtsteiner, Posch, and Delbruck 2008) (up) and then we depict the changes we suggest in red (below). The modulation is achieved by means of block A and the V_{TH} modulation with peripheral circuitry. We expect that such abrupt changes in CT cannot be made by simply modulating V_{TH}, since the voltage swing limits also the CT range. Then, since CT also depends on the capacitance value inside block A (as didactically explained by (Posch et al. 2014)), we can also "modulate" this capacitance by changing the original capacitor by two capacitances in parallel, and with one of them switched and "controlled" by the Mod signal. The inhibition can be implemented by "deviating" the reset signal sent by the arbiter to each individual pixel when a spike is sent. Typically, this reset signal closes the reset transistor and sets V_{diff} to zero, so new events can be detected. In our case, the signal coming from the arbiter will instead charge a capacitor called C_{inh}, which will hold the reset transistor closed while this capacitor is charged. This starts the inhibition state for a single pixel, since V_{diff} is permanently set to zero and the

[35] Or, in other words, that slopes are not changing during a modulation period.

pixel becomes unable to spike. This inhibition can only be removed by a global synchronous enable signal, which discharges C_{inh} for all pixels, and regardless of if they spiked or not.

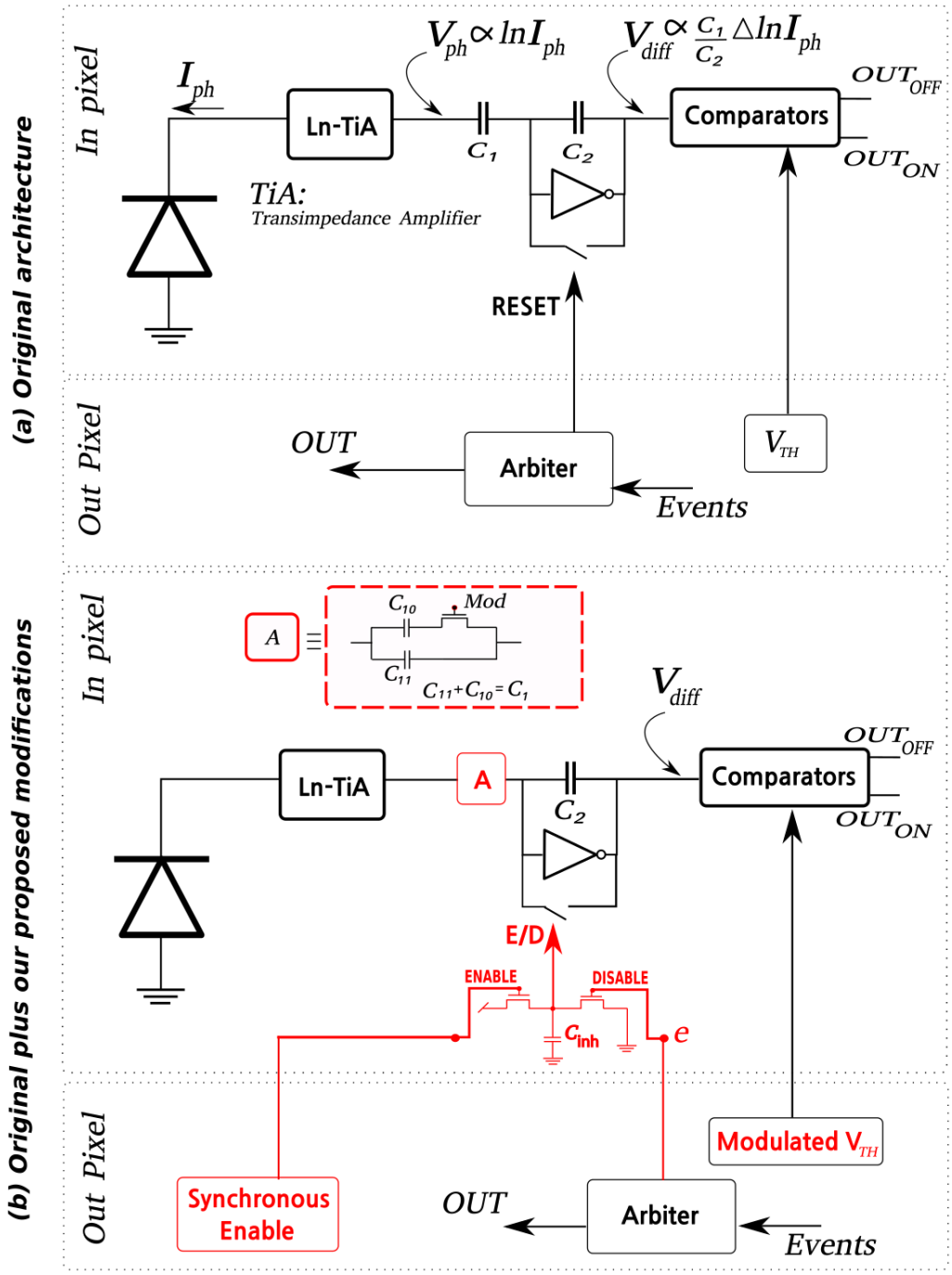


Figure 88 : DVS original schematic (a) from (Lichtsteiner, Posch, and Delbruck 2008) and (b) our proposed modification for including the modulation presented in Figure 86, and published in (Cubero et al. 2020).

Behavioral simulation results

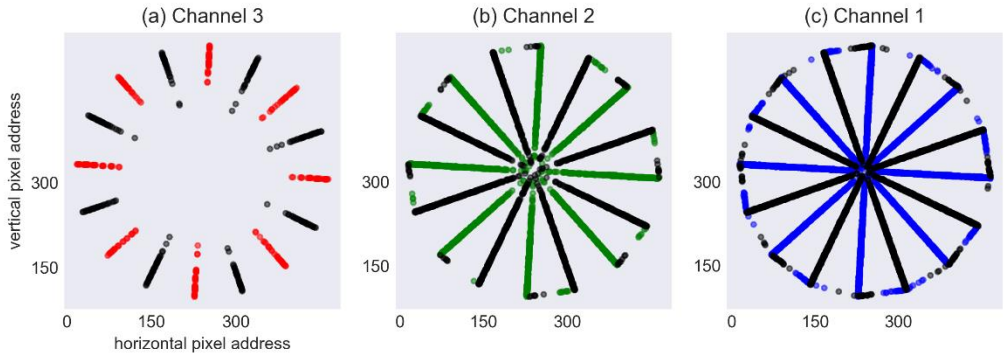


Figure 89 : example of outputs from the three channels during one modulation period (Cubero et al. 2020).

In Figure 89, we present rendered images of events corresponding to time intervals of t_l for each of the three channels. In this case, inhibition has not been applied yet. We observe that channel 3, related to the highest CT, presents events spatially located near color edges further from the center. Those locations can be related to real world particles that are moving faster under approximately uniform lighting conditions and with similar projective transformations. Moreover, channels 2 and 1, with progressively decreasing CT, start showing even near the center and the polygon perimeter. In Figure 90, we present plots of whole image data throughput vs. time for four different pre-processing schemes: standard dynamic vision sensor (DVS), the DVS with CT modulation only (mDVS), the mDVS with inhibition and synchronous reset each channel (SE-ch-mDVS), and finally the mDVS with inhibition and synchronous reset only each modulation cycle (SE-cy-mDVS).

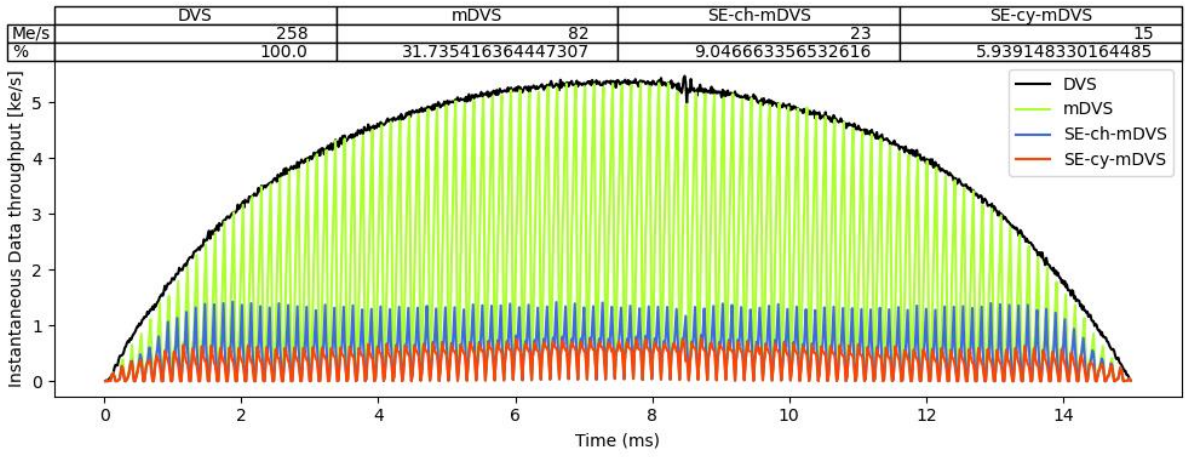


Figure 90 : instantaneous data throughput for the original DVS and for our three modifications. The table at the top presents the average data throughput.

From Figure 90, we observe that DT is reduced by applying our modulation scheme. We explain the shape of the DT-vs-time curve in terms of the polygon circular motion: the polygon starts accelerating (under a circular motion only) after begin motionless, and after it starts deaccelerating until it reaches a null angular velocity. Then, the DVS-DT curve (represented as black in last figure) may

be correlated to the polygon angular speed. The modulation (green curve) decreases DT by limiting it under a period of time for each modulation period, and thus it appears to decrease and increase abruptly while the overall curve shape follows the same as the DVS. Notice that we did not include stabilization-time-window-events in this plot. The inhibition, however, generated a sort of "flattening" or "stabilization" around a maximum DT regardless of the polygon speed. The difference between per-channel, and per-cycle synchronous enable, was the maximum settling point for the DT peaks in each period. Then, the minimum DT obtained in our simulation came from the SE-cy-mDVS, corresponding to only 5,9 % of the DT obtained from the standard DVS.

In last paragraphs we have discussed about how to reduce DT from a standard DVS by means of a convenient CT modulation. Nevertheless, this exercise only allowed us to better understand dynamic vision principles, and to observe that even after the potential DT reduction, the implementation for object detection is far from being completed. Moreover, we did not find a way which is simple enough and that allows combining functionalities of a standard DVS alongside with edge detection complying with characteristics mentioned in chapter 6 (low spatial noise, enhanced feature-DR and orientation information). Actually, besides complications due to electronic-overhead for event detection and processing, we did not find an embedded algorithm that combines ideas from both approaches. The problem stated here, is the reason why we introduce the "Dynamic Features Vision Sensor", further explained in next subsection.

7.5. Using dynamic vision for ROI proposals

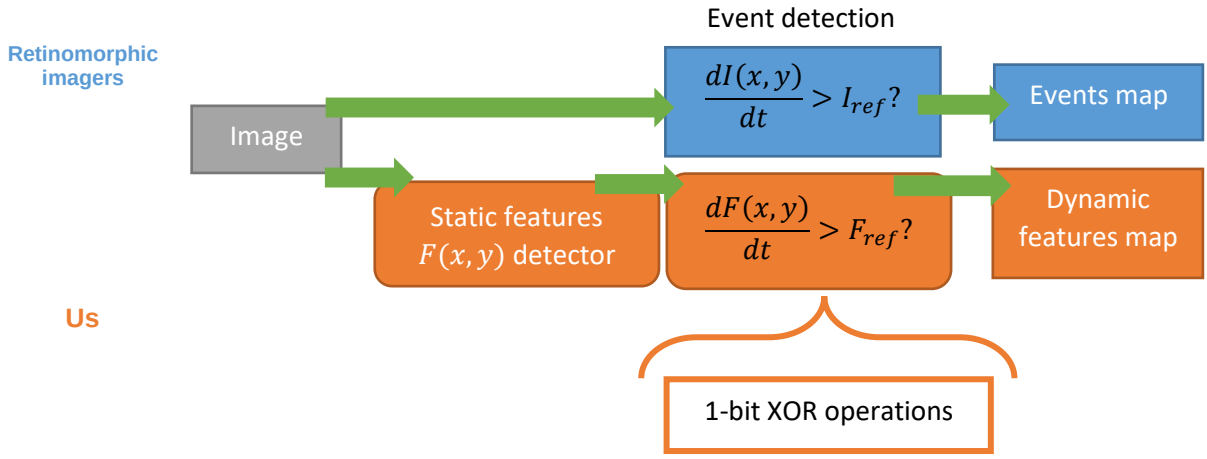


Figure 91 : our dynamic features extraction method compared with the state of the art.

Figure 91 presents a scheme of our proposed pre-processing pipeline for mixing both spatial and temporal features for low power object localization and classification. In Figure 91, the block "Image" represents the matrix of pixels. The block "static features detector" corresponds to a feature extractor such as a parallelized edge extractor. Then, event detection blocks relate to how motion is detected along the image. For a DVS, motion is detected by comparing, at pixel level, the relative light changing with a reference. In our case, we detect motion by comparing feature magnitudes codified in 1-bit per pixel. Indeed, we lay on the hypothesis that the equivalent frame-rate resulting from rendering event-frames from an asynchronous dynamic vision sensor is too high and unnecessary. The justification is that a very high frame rate also imposes a bigger load on further processing, and this

situation may not be compatible with low power applications. Then, it becomes logical to reinforce pre-processing stages in order to diminish the amount of bits passed to other processing layers. Moreover, by accepting a reduced frame rate, we suspect that the asynchronous nature of the DVS becomes less important, and that a standard frame based approach may be simpler to implement and effective for OD. Nevertheless, frame-based approaches can make challenging to recover dynamic information, contrary to a DVS approach. So, our idea to circumvent this dichotomy is to derive dynamic data from feature map instead of from the light intensity map (standard image frame). In our case, feature is a 1-bit edge, so dynamic features can be obtained with simple 1-b XOR comparisons between successive features frames, in contrast with the in-pixel light intensity change detection that would require n-bit operations. Figure 92 explains more in detail how dynamic data is obtained in our case:

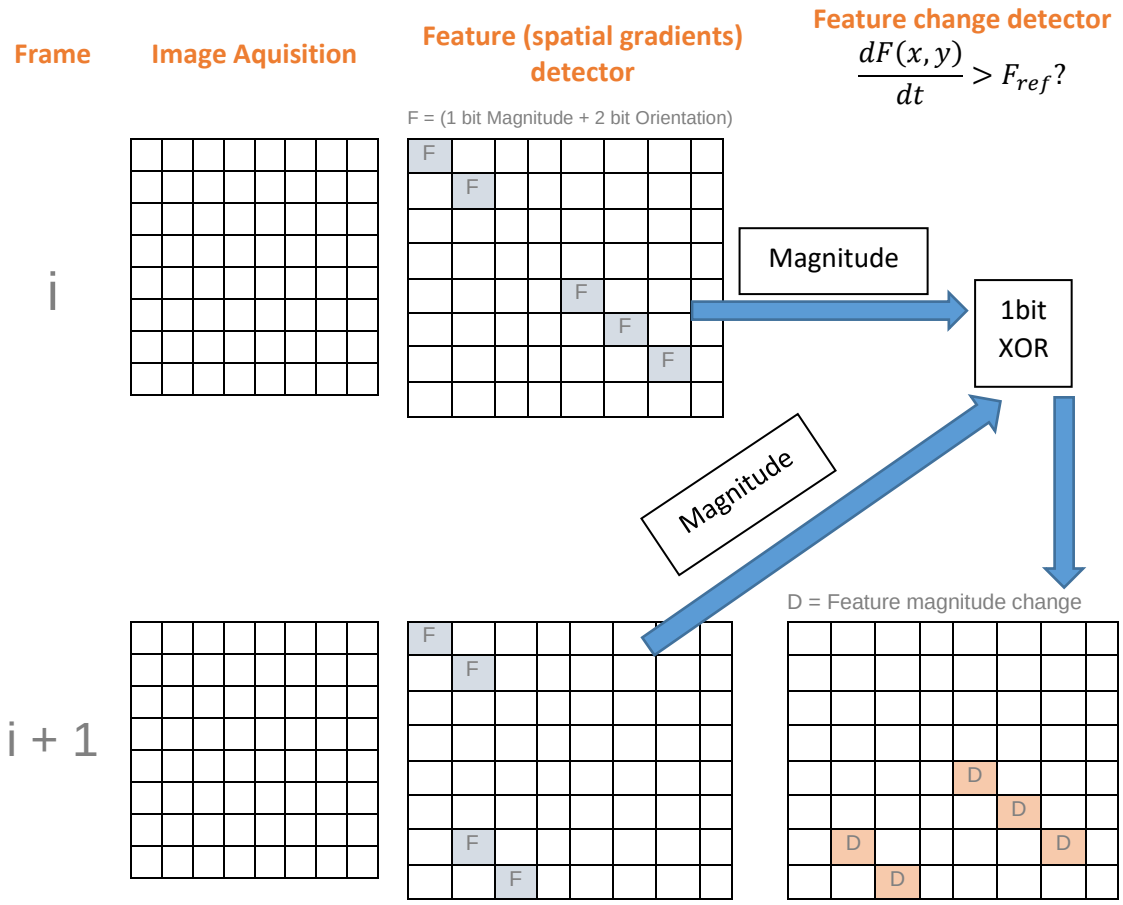


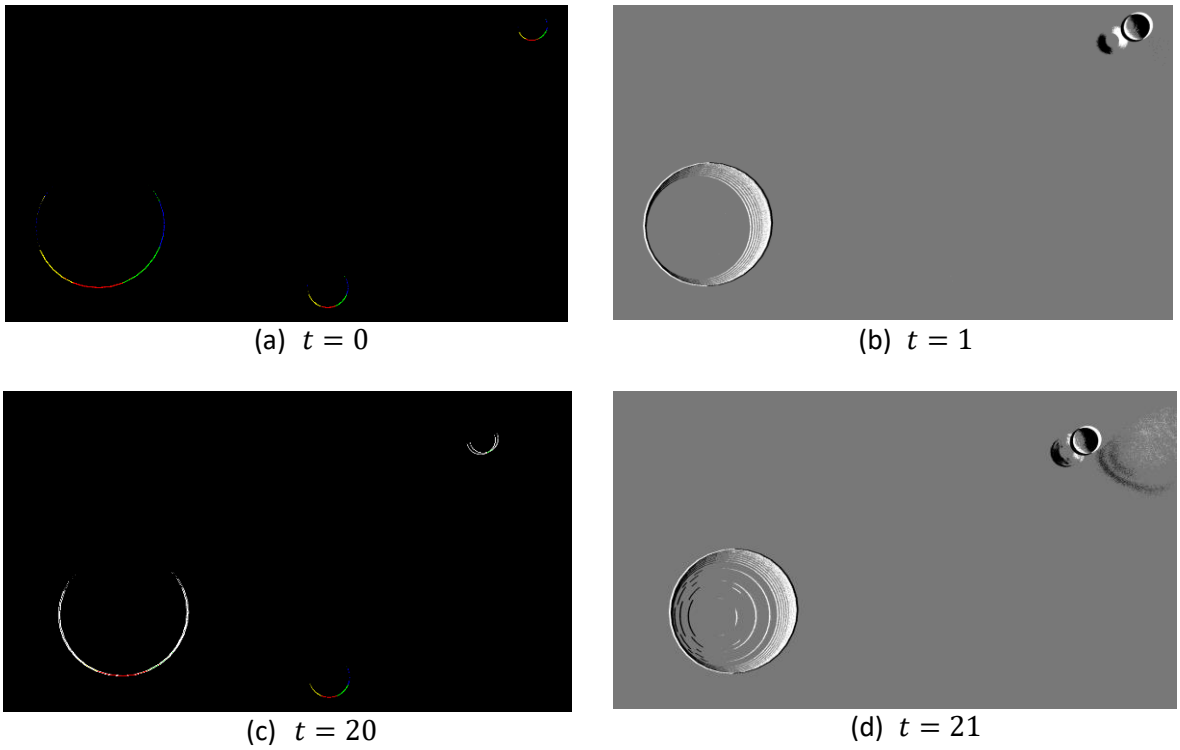
Figure 92 : diagram illustrating the principle of functioning for extracting dynamic features from two subsequent feature maps.

The scheme in Figure 92 shows the principle of the dynamic feature vision sensor: it relies on the incoming feature-magnitudes, output from the edge-detector, and the last feature-map. For the first frame after the imager is powered, or after a convenient amount of time (much higher than the frame period), the dynamic-features-map is initialized as a copy of the 1-bit feature-magnitudes-map $F_{i=0}$. Then, for each new incoming feature-magnitudes-map F_{i+1}, the new dynamic-feature-map D_{i+1} (bottom right in figure) is updated as $D_{i+1} = F_i \oplus F_{i+1}$. With "$\oplus$" representing the 1-bit XOR function spanned for every single pixel. The result is that D_j can be rendered as an image of locations where feature magnitudes changed $(1 \to 0, 0 \to 1)$. Thus, we interpret D as an approximation of the

feature magnitude derivative with respect to time. We observed some advantages of such scheme with respect to the standard DVS:

1. Since dynamic features are constrained to appear in locations where a static-spatial feature was present, dynamic outputs are sparser than DVS outputs (when thinking of rendering an image of events at the same frame rate).
2. The pre-processing allows tracking directly moving edges in the scene.
3. The feature derivative is represented by a simple 1-bit XOR operation, in contrast with the in-pixel light intensity change detection.
4. The edges output, alongside with the dynamic output, makes compatible the 4-bit output for OD with algorithms already tested in the state of the art and in this work (for object localization with Edge-Boxes with reduced bit-depth input).
5. The dynamic feature map initialization allows detecting also static objects.

For instance, in Figure 93, we present 4 different output frames obtained with EdgeTon (Cubero et al. 2019). The right side corresponds to the DFVS with relative-Origrad (with average denoising and without binning), and the right side corresponds to a DVS with CT = 3 %. The colored pixels indicate static features (edges whose magnitude is 1), and white/black pixels indicate dynamic features. Notice that the right side is at time-stamps of one frame after the left side. That is, since the simulation was configured to automatically visualize (render) frames 0 and 20 from the output, but from the way we simulate the DVS makes that the output frame 20 corresponds to the frame 21 of the dataset. Nevertheless, this time-stamp difference is not important for the conclusions we drive: firstly, the top of Figure 93 indicates that the fist DFVS output corresponds only to static features, which is not different at all to the output of a simple edge-extractor. In the DVS case, we observe only the two spheres that are moving, whereas the one in the center was not seen since it is not in motion. Then, the DFVS allows obtaining static objects by analyzing the first frame on the image as for the case of one typical edge-extractor. The bottom of Figure 93 indicates that, as expected, dynamic features (illustrated in white or black) appear only where spheres are moving. Nevertheless, the important difference to notice is the population of events in both images. The DFVS is constrained to show motion only where there is an edge, whereas the DVS outputs events coming from lighting effects such as shadows.



(a) $t = 0$ (b) $t = 1$

(c) $t = 20$ (d) $t = 21$

Figure 93 : simulation output from the DFVS (left) and a standard DVS with CT = 3 % (right), at two different times[36] of the video sequence from the three-spheres-over-ground dataset.

We also observed advantages of our scheme with respect to standard edge-detector approaches:

1. By using the edge-detector architecture presented in chapter 4, we can separate the edge-magnitude-extraction from the angular approximation. Then, there are two potential modes of functioning: one for complete oriented edges extraction, and one for only the magnitude extraction. The latter allows to obtain only edges magnitudes alongside with a potentially sparse map of dynamic features.
2. Attention based mechanisms based on dynamic data can be used directly in the frame based algorithm in order to make it more efficient. In contrast to other works which target only wake-up mechanisms like (Choi et al. 2014), which turns detection on and off depending on the scene overall motion, dynamic features can be used to discriminate boxes that do not contain relevant dynamic information (motion). That is, instead of completely turning on/off the system, we observe regions in the image where edges are moving, and then we can modify Edge-Boxes to skip scoring boxes when there are not enough features that changed respect to last time-stamp. The modified part of Edge-Boxes is presented in orange in Figure 94 (the blue part represents the state of the art):

[36] The time-stamps of the two imagers (left respect to right) are not exactly the same, those time stamps are distanced 1 frame respect to each other, since the DFVS outputs the frame 0 corresponding to the frame 0 of the dataset, while the DVS outputs the Frame 0 from the frame 1 of the dataset. The reason is that the DVS needs at least two sequential frames to generate one simulation output frame.

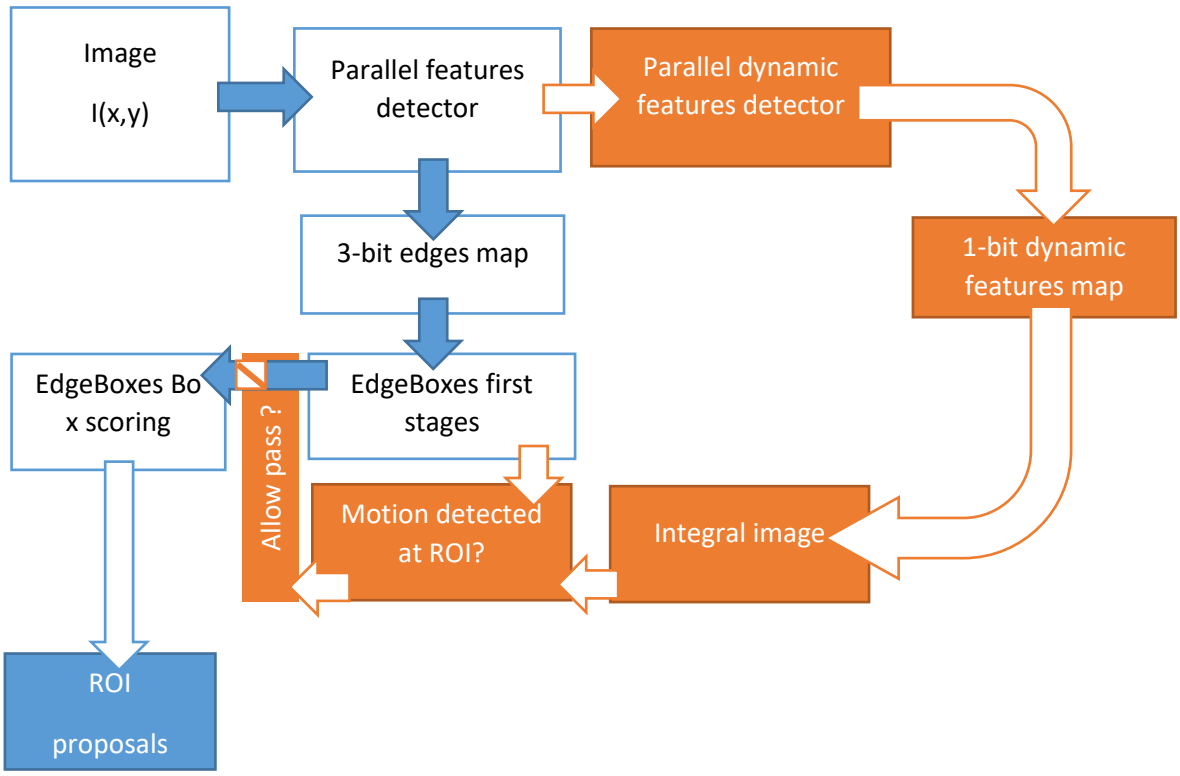


Figure 94 : diagram showing how we modified EdgeBoxes in order to include dynamic features in the ROI proposals generation.

In Figure 94, we use the dynamic features detector to obtain a 1-bit dynamic features map. We obtain the integral image of such dynamic map in order to calculate fast the amount of changing features in any hypothetical box inside Edge-Boxes. Then, for each hypothetical box, we skip the box scoring if such box does not contain an amount of changing edges higher than a threshold. If a box is not scored, then it is discarded.

Table 15 : Cummulative time in seconds per image

Architecture	Edgeboxes	Boosted EdgeBoxes	Improvement
HighRes Origrad	1,30	0,70	85%
HighRes Origrad + BINN	0,31	0,19	59%
LogHog	1,84	1,18	56%
LogHog + BINN	0,47	0,31	52%
Relative Origrad	1,55	0,86	80%
Relative Origrad + BINN	0,38	0,23	67%

Table 15 shows runtime results in a desktop pc, on the Three Spheres Over Ground dataset, for both Edge-Boxes and the Boosted Edge-Boxes version, and when taking into account different architectures. We observe a runtime improvement in all architectures, which is up to 85 % for the relative oriented edges extractor. Moreover, the impact is less important (67 % for the relative Origrad) when binning is used, which could suggest that when the image size is reduced the dynamic features are less important. Notice that this percentage is with respect to the original EdgeBoxes in the state of the art (Zitnick and Dollár 2014). In Table 16, we summarize the resolution used in this dataset:

Table 16: Summary of synthetic images resolution

	Pixel cols	Pixel rows	Resolution in MegaPixels
Original Res	1920	1080	1,17
Binned resolution	960	540	0,29

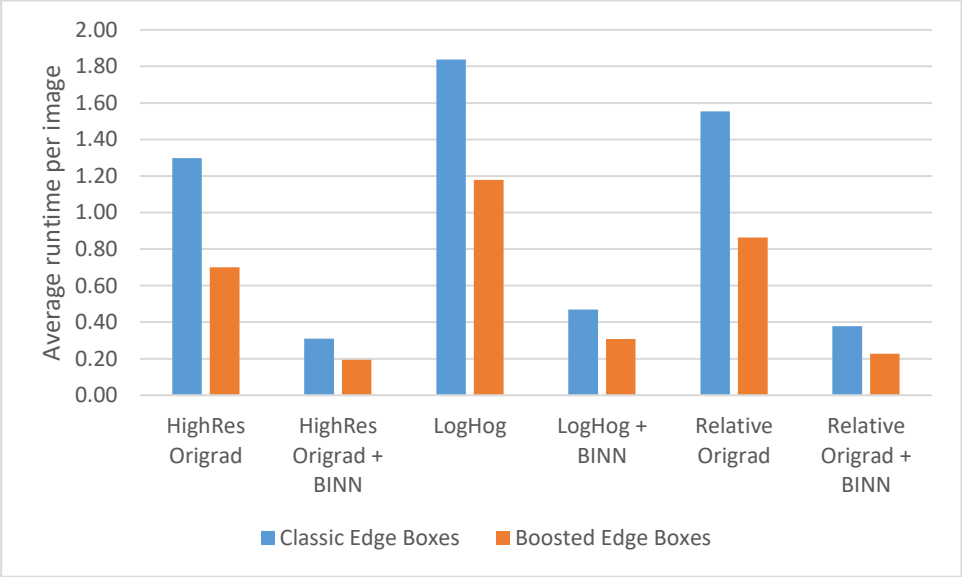


Figure 95 : average runtime per imager for different architectures, showing the gain in runtime when using dynamic features alongside with EdgeBoxes.

Figure 95 summarizes the runtimes when applying the boosting technique on EdgeBoxes, showing graphically that for the tested architectures, there was in general a gain in runtime performance. The reason for such improvement is that less boxes are tested, since they do not contain edges that are moving. We expect this method to reduce power consumption at similar runtime and detection rate, since less redundant data is processed. Of course, this reduction has to be experimentally justified, since our method also has an overhead for generating and processing the dynamic features. Respect to the localization performance, we verified that the spheres where always localized with ROIs with and IoU > 0.5, which was the case in the frames we verified.

7.6. Conclusions of chapter 7

In this chapter we have discussed the need of generating synthetic datasets for modeling non-standard smart image sensors. We explained basic requirements for such datasets, namely the simulation of light rays, high temporal resolution, and automatic labels (bounding boxes) generation. In addition, we explained our method for automatic labels generation using Blender, Python scripting, and projective geometry with the pinhole camera model. Thanks to such datasets, we propose possible improvements for Dynamic Vision Sensors aiming for low power and embedded object localization and classification. We observed that our modulation scheme significantly reduced the data throughput down to 5,9 % in our simulation respect to a standard DVS with a constant CT, and stabilized the instantaneous data throughput along time regardless of the speed of different particles in a simplistic

scene. Moreover, we observed a dichotomy between DVS schemes and frame based oriented edges extractors, from the point of view of implementation. Then, we have proposed a way to circumvent this issue by obtaining dynamic data in the digital domain (1-bit logic operations) from feature magnitudes, instead of performing in pixel lighting-derivate approximations. Our proposition drawback is the need for memorizing the last feature-magnitude map (so the required memory is of columns x rows bits). However, we showed, by means of a behavioral simulation, that our dynamic features can be successfully incorporated in more classic algorithms for object localization such as Edge-Boxes. Moreover, we found that dynamic features allow to increase localization efficiency by neglecting hypothetical boxes (ROI proposals) which do not contain enough dynamic information. For example, using our Relative Oriented Gradients architecture, we found a runtime reduction of 60 % (with binning) and 80 % (without).

Chapter 8. Conclusion

We have studied, and preliminary characterized (with behavioral simulations and on-paper electronic-models) the use of a ROI-proposals with Edge-Boxes for hardware-constrained devices. That is, in the context of integrated-IC smart-image-sensing architectures. Our motivation was to replace the typically used "sliding-window-approach" in embedded smart-image-systems scenarios. Moreover, we targeted a more general and potentially energy-efficient approach. We took also into account the potential inclusion of principles from event-based-cameras for improving the energy efficiency of ROI-proposals generation. Nevertheless, we did not rely in event-based cameras for ROI proposals, since they present problems such as less mature algorithms of event-based OD, and high pixel/read-out complexities.

In chapter 2, we have introduced the theoretical background and some state of the art examples that we found closely related with our work. We cited standard (frame-based) pixel topologies, and contributions from the SoA for event-based architectures (which capture efficiently the dynamic data). Moreover, from our survey in embedded OD, we observed that some works have embedded/adapted different kinds of OD pipelines already tested for general purpose (e.g. desktop) computing machines. There are in general two kind of approaches: some that rely in the so-called "hand-crafted-features" (typically based on light-intensity gradients, when taking into account the integrated-IC context), and others using "learned-features" (with for example quantized-convolutional-neural-networks). The point of generating features so soon is to reduce power-consumption due to image analog-to-digital conversion and high data-rates. There are also two main trends in OD for general purpose machines: firstly, the ones base on first region proposal stage, and others that compute both classification and localization in parallel along the entire image. We cited those since other research groups have taken inspiration from them as well for implementing OD in hardware-constrained contexts.

From our survey in chapter 2, we took preliminary decisions for settling the path of our work:

1. Firstly, we preferred using hand-crafted features generated near the pixels-array (instead of learned-features computation). Our justification was that hand-crafted features do not require weight-access for multiply-accumulation operations, max-pooling and activation functions so soon at the pre-processing level. Also, supervised learning (e.g. with SVM, CNNs, etc…) can be used at later stages with those hand-crafted-features. In addition, we could focus on optimizing an electronic-architecture for a specific feature type.
2. Secondly, we selected a ROI-proposals based pipeline instead of a fully parallelized one. Our motivation was that computing ROI-proposals, and then running a classifier only on those ROIs could be potentially more energy efficient than testing the classifier along many image sub-regions in parallel. In addition, the two stage approach (localization and then classification) seems easier to combine with typical wake-up approaches for power savings.
3. Thirdly, we decided to take one (often used) figure of merit in the SoA for our comparisons: "the power per pixel per frame", which represents the power consumption normalized to frame rate and number of pixels. Of course, this power consumption is related to one specific task (e.g. feature extraction, image-acquisition, classification, etc…), or a particular combination of them. This figure for merit allows comparing smart-imaging-systems regardless of if they are implemented with different pixels-amounts, and/or frame-rates.

The drawback is that it neglects some design parameters such as bandwidth (systems optimized to run at high frame-rates could potentially consume more power due to bandwidth constraints). Another drawback is that many works report this figure of merit for different tasks, which makes the comparison less evident.

4. Finally, we decided to use event-based approaches only as inspiration for improving the frame-based pipeline, since, at first glance, we did not find a particular motivation to use a fully event-based system for embedded OD. That is, one motivation that compensates the pixel/read-out/asynchronous-processing complexities.

In chapter 3, we have presented our methodology for performing behavioral simulations, and from which our conclusions are driven. Indeed, the problem we wanted to solve implied taking into account metrics and design parameters from both Machine Learning and Embedded-IC contexts. The difficulty we found is that Machine Learning tools are typically too "high level" in comparison with tools for circuit-architecture design. Then, we took that as motivation for developing our own simulation framework, which we called EdgeTon. With EdgeTon, we could take into account (behaviorally) circuit-architecture constraints on the feature extraction stage, while automatically passing the smart-image-system-output to computer-vision/machine learning algorithms (for benchmarking the AI performance). Then, we could set general design parameters for the feature-extractor (such as equivalent number of bits or ENOB for the pre-processing stage) and relate it to object-detection-related metrics such as Intersection-over-union for the localization stage.

In chapter 4, we have studied the feasibility of an implementation of ROIs-proposals with Edge-Boxes in constrained devices. For such objective, we decorticated Edge-Boxes and estimated a worst case scenario memory usage of ~25 Mbytes. Moreover, we present a possible pipeline optimized for low-level features generation for Edge-Boxes, namely the oriented edges. We proposed to obtain those oriented edges in 3 bits (1 bit for the magnitude, and 2 for the orientation). In contrast with the SoA, we propose generating high-dynamic range low-level-features but still computing the angle in a linear way, which has the advantage of generating angular information better suited for Edge-Boxes. Moreover, we propose a cascaded de-noising plus subtraction pre-processing at bottom-column, in order to diminish the number of false-edges detected, and thus significantly reducing memory and runtime footprints for Edge-Boxes execution in comparison with other embedded edge-extractors.

In chapter 5, we have introduce the low-level-features or oriented-edges extractor analog circuitry. We have divided the pre-processing in 5 main stages, for which we approximate – theoretically, on paper-, design parameters such as bandwidth and biasing current. We base our calculations on arbitrary yet illustrative values of biasing Voltage (1.5 V), matrix of pixels size (500 x 500 pixels), sampling/amplification capacitances (100 fF) and frame-rate (60 fps). Our theoretical calculation gives an ideal value of ~56.7 fJ/pixel/frame, which is ~10^3 times better in power consumption than other analog or digital low-level oriented-edge extractors we found in the SoA. We think that the reason of such result is that we do not try to achieve an unnecessary high ENOB before quantization of the low-level features, since even with a relatively low theoretical ENOB of ~8 bits before quantization, the oriented-edges are sufficient for ROI proposals, as shown in chapter 6. Moreover, our architecture exploits a dense parallelization of pre-processing without the need of any programmability, and it is based on simpler computations (averaging, and subtraction) instead of the more complex ratio to digital conversion (which implies a division-operation on the mixed-signal part).

In chapter 6, we presented benchmarks characterizing the performance of different pre-processing architectures (namely, oriented-edges extractors) and by means of the IoU. Moreover, we presented benchmarks characterizing the memory/runtime dependence of Edge-Boxes upon the edge-extractor architecture. The main result from this chapter is that we show that ROI proposals

generation with aggressively quantized low-level features is still possible, with a global average IoU (with the default Edge-Boxes parameters) higher than 0.5 and which approximates to 0.7 in some cases. We proposed a figure of merit that takes into account the global average best IoU, memory and runtime to characterize different edge-extractors performances, and we found that the best FOM corresponds to the relative Origrad with Average De-noising and 2x2 Binning.

In chapter 7, we have proposed modification to the dynamic vision sensor in order to reduce data throughput with a minimal electronics overhead. Moreover, we have proposed a way to efficiently obtain dynamic data in a frame-based manner, and we have shown how to include this dynamic data to boost the runtime performance of region proposals generation with Edge-Boxes. For instance, we found a runtime reduction of 60 % (with binning) and 80 % (without binning) when using our relative oriented gradients architecture (with respect to not using dynamic data for ROI-proposals). Moreover, we have presented a programmatic approach to generate labeled datasets for supervised learning and object detection in Blender.

This thesis has explored, at behavioral simulation level, the inclusion of a Region Proposals algorithm in embedded smart imagers. Nevertheless, the path we took for approaching the subject let still many points that can be addressed by further works. One idea is to continue this thesis by targeting the Edge-Boxes digital implementation. We believe that at least some stages of Edge-Boxes can be "rethought" and parallelized (but keeping the essential ideas), which could significantly speed-up the ROI proposals generation. Another idea to explore is reducing Edge-Boxes bit-depth computations in order to reduce the memory required. In addition, it could be interesting to explore the Faster-RCNN pipeline in an embedded context, which uses CNN-like features for ROI-proposals (and thus for replacing Edge-Boxes with a CNN in an embedded context). One possible way of doing that could be by using the edge-map output from Origrad as input to a digital CNN which generates a feature map from low bit-depth hand-crafted features, and not directly from light intensity. Furthermore, generating an edge-map from each color channel could present advantages for recovering color-information in the edge-map, and potentially improving localization/classification performance. Finally, our approach used in chapter 7, of boosting runtime of ROIs generation with dynamic data, could be potentially used with CNN features or other kinds of features as well, thus opening doors for exploring more elaborated algorithms for low-power object detection.

References

Acharya, Jyotibdha, Vandana Padala, and Arindam Basu. 2019. “Spiking Neural Network Based Region Proposal Networks for Neuromorphic Vision Sensors.” In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, 1–5. Sapporo, Japan: IEEE. https://doi.org/10.1109/ISCAS.2019.8702651.

Alexe, B., T. Deselaers, and V. Ferrari. 2012. “Measuring the Objectness of Image Windows.” *IEEE Transactions on Pattern Analysis and Machine Intelligence* 34 (11): 2189–2202. https://doi.org/10.1109/TPAMI.2012.28.

Barchid, Sami, Jose Mennesson, and Chaabane Djeraba. 2021. “Deep Spiking Convolutional Neural Network for Single Object Localization Based On Deep Continuous Local Learning.” In *2021 International Conference on Content-Based Multimedia Indexing (CBMI)*, 1–5. Lille, France: IEEE. https://doi.org/10.1109/CBMI50038.2021.9461880.

Berner, Raphael, Christian Brandli, Minhao Yang, Shih-Chii Liu, and Tobi Delbruck. 2013. “A 240x180 10mW 12us Latency Sparse-Output Vision Sensor for Mobile Applications.” 2013 Symposium on VLSI Circuits, pp. C186–C187.

“Blender.” n.d. Blender. Accessed July 20, 2021. https://www.blender.org/.

“Blender 2.93 Manual, Cycles.” 2021. Blender 2.93 Manual. July 20, 2021. https://docs.blender.org/manual/en/latest/render/cycles/index.html.

Bose, Laurie, Jianing Chen, Stephen J. Carey, Piotr Dudek, and Walterio Mayol-Cuevas. 2019. “A Camera That CNNs: Towards Embedded Neural Networks on Pixel Processor Arrays.” *ArXiv:1909.05647 [Cs]*, September. http://arxiv.org/abs/1909.05647.

“Box Plot.” 2021. Wikipedia. July 7, 2021. https://en.wikipedia.org/wiki/Box_plot.

“Camera Calibration.” 2021. Documentation. OpenCV, Open Source Computer Vision, 4.5.3-Dev. June 8, 2021. https://docs.opencv.org/master/dc/dbb/tutorial_py_calibration.html.

Canny, John. 1986. “A Computational Approach to Edge Detection.” *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8 (6): 679–98. https://doi.org/10.1109/TPAMI.1986.4767851.

Choi, Jaehyuk, Seokjun Park, Jihyun Cho, and Euisik Yoon. 2014. “A 3.4-μW Object-Adaptive CMOS Image Sensor With Embedded Feature Extraction Algorithm for Motion-Triggered Object-of-Interest Imaging.” *IEEE Journal of Solid-State Circuits* 49 (1): 289–300. https://doi.org/10.1109/JSSC.2013.2284350.

Cubero, Luis, Arnaud Peizerat, Dominique Morche, and Gilles Sicard. 2019. “Smart Imagers Modeling and Optimization Framework for Embedded AI Applications.” In *2019 15th Conference on Ph.D Research in Microelectronics and Electronics (PRIME)*, 245–48. Lausanne, Switzerland: IEEE. https://doi.org/10.1109/PRIME.2019.8787750.

Cubero, Luis, Arnaud Peizerat, Dominique Morche, and Gilles Sicard. 2020. “Event Threshold Modulation in Dynamic Vision Spiking Imagers for Data Throughput Reduction.” *Electronic Imaging* 2020 (7): 145-1-145–46. https://doi.org/10.2352/ISSN.2470-1173.2020.7.ISS-145.

Dalal, N., and B. Triggs. 2005. “Histograms of Oriented Gradients for Human Detection.” In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05)*, 1:886–93. San Diego, CA, USA: IEEE. https://doi.org/10.1109/CVPR.2005.177.

Delbruck, Tobi, and Raphael Berner. 2010. “Temporal Contrast AER Pixel with 0.3%-Contrast Event Threshold.” In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, 2442–45. Paris, France: IEEE. https://doi.org/10.1109/ISCAS.2010.5537153.

Deng, Yongjian, Hao Chen, and Youfu Li. 2021. “MVF-Net: A Multi-View Fusion Network for Event-Based Object Classification.” *IEEE Transactions on Circuits and Systems for Video Technology*, 1–1. https://doi.org/10.1109/TCSVT.2021.3073673.

Deng, Yongjian, Youfu Li, and Hao Chen. 2020. “AMAE: Adaptive Motion-Agnostic Encoder for Event-Based Object Classification.” *IEEE Robotics and Automation Letters* 5 (3): 4596–4603. https://doi.org/10.1109/LRA.2020.3002480.

Dollar, Piotr, and C. Lawrence Zitnick. 2015a. “Fast Edge Detection Using Structured Forests.” *IEEE Transactions on Pattern Analysis and Machine Intelligence* 37 (8): 1558–70. https://doi.org/10.1109/TPAMI.2014.2377715.

Dollar, Piotr, and Larry Zitnick. 2015b. “Edge Boxes Source Code.” Code repository. Github. February 15, 2015. https://github.com/pdollar/edges/blob/master/private/edgeBoxesMex.cpp.

Eki, Ryoji, Satoshi Yamada, Hiroyuki Ozawa, Hitoshi Kai, Kazuyuki Okuike, Hareesh Gowtham, Hidetomo Nakanishi, et al. 2021. “9.6 A 1/2.3inch 12.3Mpixel with On-Chip 4.97TOPS/W CNN Processor Back-Illuminated Stacked CMOS Image Sensor.” In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, 154–56. San Francisco, CA, USA: IEEE. https://doi.org/10.1109/ISSCC42613.2021.9365965.

El Gamal, A., and H. Eltoukhy. 2005. “CMOS Image Sensors.” *IEEE Circuits and Devices Magazine* 21 (3): 6–20. https://doi.org/10.1109/MCD.2005.1438751.

Everingham, M., L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. n.d. “The PASCAL Visual Object Classes Challenge 2007.” The PASCAL Visual Object Detection Challenge 2007. Accessed July 26, 2021a. http://host.robots.ox.ac.uk/pascal/VOC/voc2007/.

———. n.d. “The PASCAL Visual Object Classes Challenge 2007 Results.” PASCALNETWORK.ORG. Accessed July 26, 2021b. http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html.

Everingham, Mark, S. M. Ali Eslami, Luc Van Gool, Christopher K. I. Williams, John Winn, and Andrew Zisserman. 2015. “The Pascal Visual Object Classes Challenge: A Retrospective.” *International Journal of Computer Vision* 111 (1): 98–136. https://doi.org/10.1007/s11263-014-0733-5.

Felzenszwalb, P F, R B Girshick, D McAllester, and D Ramanan. 2010. “Object Detection with Discriminatively Trained Part-Based Models.” *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32 (9): 1627–45. https://doi.org/10.1109/TPAMI.2009.167.

Felzenszwalb, Pedro, David McAllester, and Deva Ramanan. 2008. “A Discriminatively Trained, Multiscale, Deformable Part Model.” In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, 1–8. Anchorage, AK, USA: IEEE. https://doi.org/10.1109/CVPR.2008.4587597.

Girshick, Ross. 2015. “Fast R-CNN.” In *2015 IEEE International Conference on Computer Vision (ICCV)*, 1440–48. Santiago, Chile: IEEE. https://doi.org/10.1109/ICCV.2015.169.

Gori, Marco. 2018. “What’s Wrong with Computer Vision?” In *Artificial Neural Networks in Pattern Recognition*, 3–16. Lecture Notes in Computer Science. Siena, Italy: Springer.

Gottardi, Massimo, and Michela Lecca. 2019. “A 64×64 Pixel Vision Sensor for Local Binary Pattern Computation.” *IEEE Transactions on Circuits and Systems I: Regular Papers* 66 (5): 1831–39. https://doi.org/10.1109/TCSI.2018.2883792.

Guo, Xiaochuan, Xin Qi, and John G. Harris. 2007. “A Time-to-First-Spike CMOS Image Sensor.” *IEEE Sensors Journal* 7 (8): 1165–75. https://doi.org/10.1109/JSEN.2007.900937.

He, Kaiming, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2018. “Mask R-CNN.” *ArXiv:1703.06870 [Cs]*, January. http://arxiv.org/abs/1703.06870.

Hsu, Tzu-Hsiang, Yen-Kai Chen, Tai-Hsing Wen, Wei-Chen Wei, Yi-Ren Chen, Fu-Chun Chang, Hyunjoon Kim, et al. 2019. “A 0.5V Real-Time Computational CMOS Image Sensor with Programmable Kernel for Always-On Feature Extraction.” In *2019 IEEE Asian Solid-State Circuits Conference (A-SSCC)*, 33–34. Macau, Macao: IEEE. https://doi.org/10.1109/A-SSCC47793.2019.9056945.

Hsu, Tzu-Hsiang, Yi-Ren Chen, Ren-Shuo Liu, Chung-Chuan Lo, Kea-Tiong Tang, Meng-Fan Chang, and Chih-Cheng Hsieh. 2021. “A 0.5-V Real-Time Computational CMOS Image Sensor With Programmable Kernel for Feature Extraction.” *IEEE Journal of Solid-State Circuits* 56 (5): 1588–96. https://doi.org/10.1109/JSSC.2020.3034192.

"Jaccard Index." 2021. Wikipedia, The Free Encyclopedia. August 27, 2021. https://en.wikipedia.org/wiki/Jaccard_index.

Jin, Minhyun, Hyeonseob Noh, Minkyu Song, and Soo Youn Kim. 2020. "Design of an Edge-Detection CMOS Image Sensor with Built-in Mask Circuits." *Sensors* 20 (13): 3649. https://doi.org/10.3390/s20133649.

Khan, Salman, Hossein Rahmani, Syed Afaq Ali Shah, and Mohammed Bennamoun. 2018. *A Guide to Convolutional Neural Networks for Computer Vision*. Synthesis Lectures On Computer Vision. Morgan & Claypool.

Kim, Hyeon-June, Sun-Il Hwang, Jae-Hyun Chung, Jong-Ho Park, and Seung-Tak Ryu. 2017. "A Dual-Imaging Speed-Enhanced CMOS Image Sensor for Real-Time Edge Image Extraction." *IEEE Journal of Solid-State Circuits* 52 (9): 2488–97. https://doi.org/10.1109/JSSC.2017.2718665.

Kim, Ji-Hoon, Changhyeon Kim, Kwantae Kim, and Hoi-Jun Yoo. 2019. "An Ultra-Low-Power Analog-Digital Hybrid CNN Face Recognition Processor Integrated with a CIS for Always-on Mobile Devices." In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, 1–5. Sapporo, Japan: IEEE. https://doi.org/10.1109/ISCAS.2019.8702698.

Kim, Seijoon, Seongsik Park, Byunggook Na, and Sungroh Yoon. 2019. "Spiking-YOLO: Spiking Neural Network for Energy-Efficient Object Detection." *ArXiv:1903.06530 [Cs, Stat]*, November. http://arxiv.org/abs/1903.06530.

Lamba, Sahil, and Rishab Lamba. 2019. "Spiking Neural Networks Vs Convolutional Neural Networks for Supervised Learning." In *2019 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS)*, 15–19. Greater Noida, India: IEEE. https://doi.org/10.1109/ICCCIS48478.2019.8974507.

Lee, Changhyuk, Wei Chao, Sunwoo Lee, James Hone, Alyosha Molnar, and Sang Hoon Hong. 2015. "A Low-Power Edge Detection Image Sensor Based on Parallel Digital Pulse Computation." *IEEE Transactions on Circuits and Systems II: Express Briefs* 62 (11): 1043–47. https://doi.org/10.1109/TCSII.2015.2455354.

Lee, Juseong, Hoyoung Tang, and Jongsun Park. 2018. "Energy Efficient Canny Edge Detector for Advanced Mobile Vision Applications." *IEEE Transactions on Circuits and Systems for Video Technology* 28 (4): 1037–46. https://doi.org/10.1109/TCSVT.2016.2640038.

Li, Chenghan, Luca Longinotti, Federico Corradi, and Tobi Delbruck. 2019. "A 132 by 104 10ʒm-Pixel 250ʒW 1kefps Dynamic Vision Sensor with Pixel-Parallel Noise and Spatial Redundancy Suppression," 2.

Lichtsteiner, Patrick, Christoph Posch, and Tobi Delbruck. 2008. "A 128x128 120 DB 15 µs Latency Asynchronous Temporal Contrast Vision Sensor." *IEEE Journal of Solid-State Circuits* 43 (2): 566–76. https://doi.org/10.1109/JSSC.2007.914337.

Liu, Wei, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. 2016. "SSD: Single Shot MultiBox Detector." *ArXiv:1512.02325 [Cs]* 9905: 21–37. https://doi.org/10.1007/978-3-319-46448-0_2.

Maloberti, Franco. 2007. *Data Converters*. Springer.

Millet, L., S. Chevobbe, C. Andriamisaina, E. Beigne, F. Guellec, T. Dombek, L. Benaissa, et al. 2018. "A 5500FPS 85GOPS/W 3D Stacked BSI Vision Chip Based on Parallel in-Focal-Plane Acquisition and Processing." In *2018 IEEE Symposium on VLSI Circuits*, 245–46. Honolulu, HI: IEEE. https://doi.org/10.1109/VLSIC.2018.8502290.

Omid-Zohoor, Alex, Christopher Young, David Ta, and Boris Murmann. 2018. "Toward Always-On Mobile Object Detection: Energy Versus Performance Tradeoffs for Embedded HOG Feature Extraction." *IEEE Transactions on Circuits and Systems for Video Technology* 28 (5): 1102–15. https://doi.org/10.1109/TCSVT.2017.2653187.

"OpenCV, Canny Edge Detector." 2021. Documentation. OpenCV, Open Source Computer Vision. July 21, 2021. https://docs.opencv.org/3.4/da/d5c/tutorial_canny_detector.html.

"OpenCV, EdgeBoxes." 2021. Documentation. Open Source Computer Vision (OpenCV). July 21, 2021. https://docs.opencv.org/master/d4/d0d/group__ximgproc__edgeboxes.html#ga226d8e1a2dee5dfb58708b97f490b18a.

"OpenCV, Image Filtering." 2021. Documentation. Open Source Computer Vision (OpenCV). July 22, 2021. https://docs.opencv.org/3.4/d4/d86/group__imgproc__filter.html.

OpenCV-dev-team. 2012. "Camera Calibration and 3D Reconstruction." OpenCV 2.3.2 Documentation. March 30, 2012. http://www.opencv.org.cn/opencvdoc/2.3.2/html/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html?highlight=findfun.

"Pixel Connectivity." 2019. Wikipedia, The Free Encyclopedia. September 17, 2019. https://en.wikipedia.org/wiki/Pixel_connectivity.

Posch, Christoph, Teresa Serrano-Gotarredona, Bernabe Linares-Barranco, and Tobi Delbruck. 2014. "Retinomorphic Event-Based Vision Sensors: Bioinspired Cameras With Spiking Output." *Proceedings of the IEEE* 102 (10): 1470–84. https://doi.org/10.1109/JPROC.2014.2346153.

"Pyramid (Image Processing)." 2021. Wikipedia, The Free Encyclopedia. June 4, 2021. https://en.wikipedia.org/wiki/Pyramid_%28image_processing%29.

Ravinuthula, V., and J.G. Harris. 2004. "Time-Based Arithmetic Using Step Functions." In *2004 IEEE International Symposium on Circuits and Systems (IEEE Cat. No.04CH37512)*, I-305-I–308. Vancouver, BC, Canada: IEEE. https://doi.org/10.1109/ISCAS.2004.1328192.

Razavi, Behzad. 2001. *Design of Analog CMOS Integrated Circuits*. International Edition 2001. McGraw-Hill.

Redmon, Joseph, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. "You Only Look Once: Unified, Real-Time Object Detection." In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 779–88. Las Vegas, NV, USA: IEEE. https://doi.org/10.1109/CVPR.2016.91.

Ren, Shaoqing, Kaiming He, Ross Girshick, and Jian Sun. 2017. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39 (6): 1137–49. https://doi.org/10.1109/TPAMI.2016.2577031.

"Scale Space." 2021. Wikipedia, The Free Encyclopedia. June 1, 2021. https://en.wikipedia.org/wiki/Scale_space.

Soell, C., L. Shi, J. Roeber, M. Reichenbach, R. Weigel, and A. Hagelauer. 2016. "Low-Power Analog Smart Camera Sensor for Edge Detection." In *2016 IEEE International Conference on Image Processing (ICIP)*, 4408–12. Phoenix, AZ, USA: IEEE. https://doi.org/10.1109/ICIP.2016.7533193.

Suarez, Manuel, Víctor M. Brea, J. Fernandez-Berni, R. Carmona-Galan, G. Linan, D. Cabello, and Ángel Rodriguez-Vazquez. 2012. "CMOS-3D Smart Imager Architectures for Feature Detection." *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 2 (4): 723–36. https://doi.org/10.1109/JETCAS.2012.2223552.

Suleiman, Amr, and Vivienne Sze. 2014. "Energy-Efficient HOG-Based Object Detection at 1080HD 60 Fps with Multi-Scale Support." In *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, 1–6. Belfast, United Kingdom: IEEE. https://doi.org/10.1109/SiPS.2014.6986096.

Takayanagi, Isao, and Junichi Nakamura. 2013. "High-Resolution CMOS Video Image Sensors." *Proceedings of the IEEE* 101 (1): 61–73. https://doi.org/10.1109/JPROC.2011.2178569.

Theuwissen, Albert. 2007. "CMOS Image Sensors : State-Of-The-Art and Future Perspectives." ESSDERC 2007 - 37th European Solid State Device Research Conference, pp. 21–27. https://doi.org/10.1109/ESSDERC.2007.4430875.

Valenzuela, Wladimir, Javier E. Soto, Payman Zarkesh-Ha, and Miguel Figueroa. 2021. "Face Recognition on a Smart Image Sensor Using Local Gradients." *Sensors* 21 (9): 2901. https://doi.org/10.3390/s21092901.

Verdant, Arnaud, William Guicquero, Nicolas Royer, Guillaume Moritz, Sebastien Martin, Florent Lepin, Sylvain Choisnet, et al. 2020. "A 3.0µW@5fps QQVGA Self-Controlled Wake-Up Imager with On-Chip Motion Detection, Auto-Exposure and Object Recognition." In *2020 IEEE Symposium on VLSI Circuits*, 1–2. Honolulu, HI, USA: IEEE. https://doi.org/10.1109/VLSICircuits18222.2020.9162854.

Vivet, Pascal, Gilles Sicard, Laurent Millet, Stephane Chevobbe, Karim Ben Chehida, Luis Angel Cubero, Monte Alegre, et al. 2019. “Advanced 3D Technologies and Architectures for 3D Smart Image Sensors.” In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 674–79. https://doi.org/10.23919/DATE.2019.8714886.

Wannamaker, R.A., S.P. Lipshitz, J. Vanderkooy, and J.N. Wright. 2000. “A Theory of Nonsubtractive Dither.” *IEEE Transactions on Signal Processing* 48 (2): 499–516. https://doi.org/10.1109/78.823976.

Yang, Minhao, Shih-Chii Liu, and Tobi Delbruck. 2015. “A Dynamic Vision Sensor With 1% Temporal Contrast Sensitivity and In-Pixel Asynchronous Delta Modulator for Event Encoding.” *IEEE Journal of Solid-State Circuits* 50 (9): 2149–60. https://doi.org/10.1109/JSSC.2015.2425886.

Yin, Chin, Chin-Fong Chiu, and Chih-Cheng Hsieh. 2016. “A 0.5 V, 14.28-Kframes/s, 96.7-DB Smart Image Sensor With Array-Level Image Signal Processing for IoT Applications.” *IEEE Transactions on Electron Devices* 63 (3): 1134–40. https://doi.org/10.1109/TED.2016.2521168.

Young, Christopher, Alex Omid-Zohoor, Pedram Lajevardi, and Boris Murmann. 2019. “A Data-Compressive 1.5/2.75-Bit Log-Gradient QVGA Image Sensor With Multi-Scale Readout for Always-On Object Detection.” *IEEE Journal of Solid-State Circuits* 54 (11): 2932–46. https://doi.org/10.1109/JSSC.2019.2937437.

Zitnick, C. Lawrence, and Piotr Dollár. 2014. “Edge Boxes: Locating Object Proposals from Edges.” In *Computer Vision – ECCV 2014*, edited by David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, 8693:391–405. Lecture Notes in Computer Science. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-10602-1_26.

List of Publications

Cubero, Luis, Arnaud Peizerat, Dominique Morche, and Gilles Sicard. 2019. “Smart Imagers Modeling and Optimization Framework for Embedded AI Applications.” In *2019 15th Conference on Ph.D Research in Microelectronics and Electronics (PRIME)*, 245–48. Lausanne, Switzerland: IEEE. https://doi.org/10.1109/PRIME.2019.8787750.

Cubero, Luis, Arnaud Peizerat, Dominique Morche, and Gilles Sicard. 2020. “Event Threshold Modulation in Dynamic Vision Spiking Imagers for Data Throughput Reduction.” *Electronic Imaging* 2020 (7): 145-1-145–46. https://doi.org/10.2352/ISSN.2470-1173.2020.7.ISS-145.

Vivet, Pascal, Gilles Sicard, Laurent Millet, Stephane Chevobbe, Karim Ben Chehida, Luis Angel Cubero, Monte Alegre, et al. 2019. “Advanced 3D Technologies and Architectures for 3D Smart Image Sensors.” In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 674–79. https://doi.org/10.23919/DATE.2019.8714886.

Patents

(Request deposited, and answer pending) Cubero, Luis. (2021). “CMOS IMAGING PRE-PROCESSING FOR LOW POWER DYNAMIC AND STATIC FEATURES FOR ARTIFICIAL INTELLIGENCE”.