Supplementary Document for "SoDaCam: Software-defined Cameras via Single-Photon Imaging"

Varun Sundar[†] vsundar4@cs.wisc.edu
Andrei Ardelean[‡] a.ardelean@epfl.ch
Tristan Swedish[§] tristan@ubicept.com
Claudio Bruschini[‡] Edoardo Charbon[‡] {claudio.bruschini, edoardo.charbon}@epfl.ch
Mohit Gupta[†,§] mohitg@cs.wisc.edu

[†]University of Wisconsin-Madison [‡]École polytechnique fédérale de Lausanne [§]Ubicept

Supplementary Note 1. Video Compressive Sensing

In this supplementary note, we provide the psuedocode that describes the emulation of two- and multi-bucket cameras and mathematically describe their multiplexing masks. We also specify algorithmic details for video recovery from compressive measurements.

Multi-Bucket Capture Pseudocode

Algorithm 1 describes the emulation of J-bucket captures, denoted as $\mathcal{I}^j_{\text{coded}}(\mathbf{x})$, from the photon-cube $B_t(\mathbf{x})$ using multiplexing codes $C^j_t(\mathbf{x})$, where $1 \leq j \leq J$. Both single compressive snapshots (or one-bucket captures) and two-bucket captures can be emulated as special cases of Algorithm 1, with $J = 1$ and $J = 2$ respectively.

Algorithm 1 Multi-Bucket Capture Emulation

Require: Photon-cube $B_t(\mathbf{x})$
Number of buckets J
Multiplexing code for j^{th} bucket, $1 \leq j \leq J$, $C^j_t(\mathbf{x})$
Pixel locations $\mathcal{X}$
Total bit-planes T
Ensure: Multiplexed captures $\mathcal{I}^j_{\text{coded}}(\mathbf{x})$
function MULTIBUCKETEMULATION($B_t(\mathbf{x})$, $C^j_t(\mathbf{x})$)
$Y^j(\mathbf{x}) \leftarrow 0, \forall\, j$
for $\mathbf{x} \in \mathcal{X}, 1 \leq j \leq J$ **do**
for $1 \leq t \leq T$ **do**
$\mathcal{I}^j_{\text{coded}}(\mathbf{x}) \leftarrow \mathcal{I}^j_{\text{coded}}(\mathbf{x}) + B_t(\mathbf{x}) \cdot C^j_t(\mathbf{x})$
end for
end for
return $\mathcal{I}^j_{\text{coded}}(\mathbf{x})$
end function

Mask sequences for video compressive sensing. For a single compressive capture ($J = 1$), a sequence of binary random is used, i.e, $C^1_t(\mathbf{x}) = 1$ with probability 0.5. For a two bucket capture, we use

$$C^2_t(\mathbf{x}) = 1 - C^1_t(\mathbf{x}),$$

which is the complementary mask sequence. For $J > 2$, at each timestep t and pixel location $\mathbf{x}$, the active bucket is chosen at random:

$$C^j_t(\mathbf{x}) \leftarrow 1,\ j \sim \text{Uniform}(1, J).$$

This is a direct generalization of the masking used for both one- and two-bucket captures.

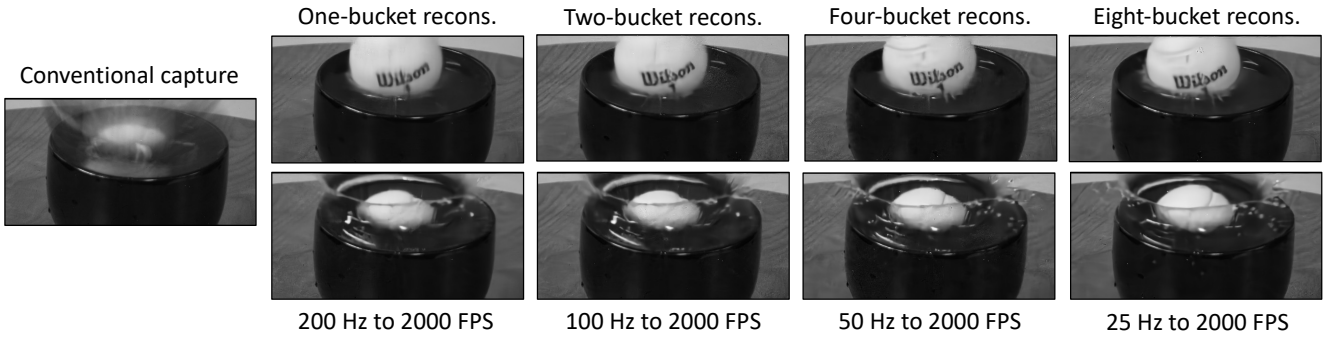
Decoding Video Compressive Captures

A variety of decoding algorithms have been developed for video compressive sensing, including: (a) optimization frameworks tailored to the forward model of Eq. (4) and with additional regularization [10, 24], (b) end-to-end deep-learning methods that utilize a large corpus of training data [7, 9, 14, 22, 23], and (c) hybrid, plug-and-play (PnP) approaches that utilize an optimization framework but perform one or more steps using a deep denoiser [5, 20, 26]. We opt to use the PnP approach featuring an ADMM formulation [25] and a deep video denoiser (FastDVDNet [16]) in this work. We justify our choice by noting that PnP-ADMM can produce high-quality reconstructions, comparable to end-to-end counterparts, while using an off-the-shelf denoiser—precluding the need to train separate models for various masking strategies.

For computational efficiency, PnP-ADMM requires the gram matrix of the resulting linear forward model of Eq. (4) to be efficiently invertible. The multi-bucket scheme described above adheres to this consideration.

Constant-Bandwidth Comparison

We now present a comparison of single-, two- and multi-bucket compressive captures when the readout rate is fixed. As Suppl. Fig. 1 shows, multi-bucket captures provide higher fidelity reconstructions even when bandwidth is fixed. Furthermore, their bandwidth cost can be amortized, to some extent, by coding only dynamic regions—we describe this next.



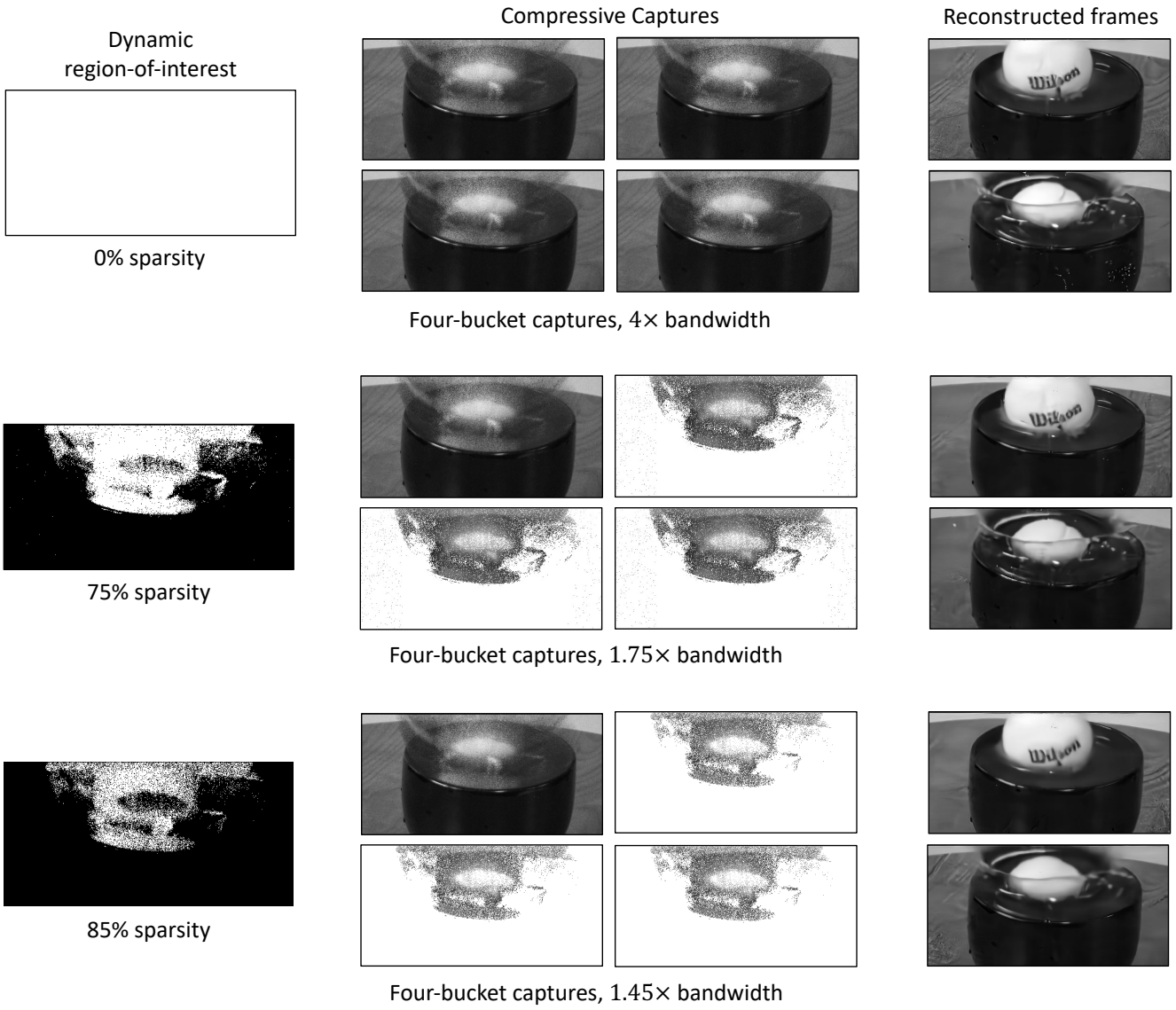
Supplementary Figure 1: **Fixed readout comparison of compressive video schemes.** We compare video reconstruction obtained from a single compressive snapshot, two-bucket capture, four-bucket capture and eight-bucket capture while holding readout constant—we achieve this by commesurately increasing readout, for instance, by reading out single compressive snapshots at 200 Hz. We indicate the readout rate here in Hertz (Hz) and the frame-rate of the reconstructed video in FPS. Clearly, multi-bucket captures provide better reconstruction results than a burst of independently multiplexed captures.

Coding Only Dynamic Regions: Mitigating the Bandwidth Cost of Multi-Bucket Captures

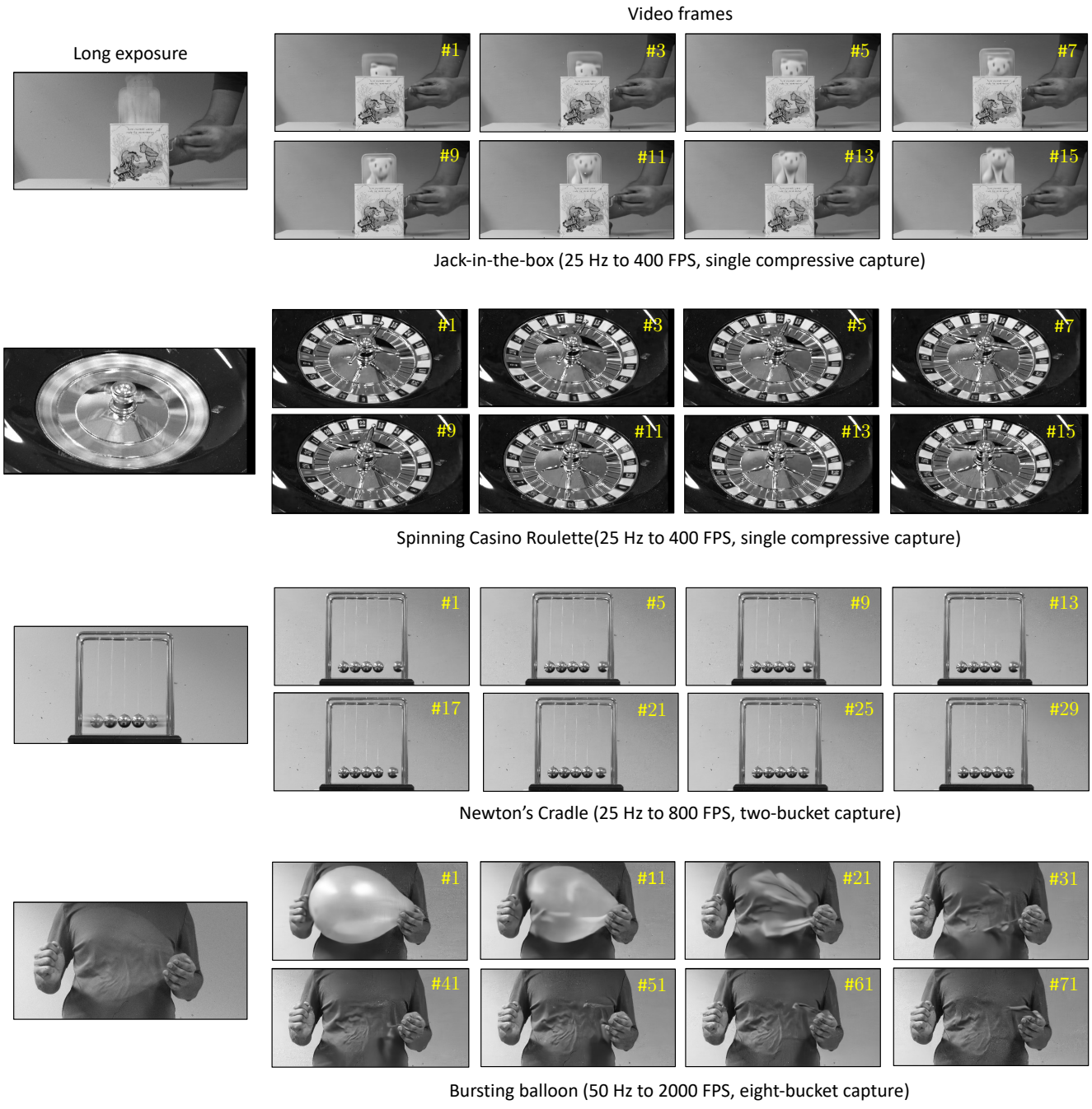
We observe that multi-bucket captures capture redundant information in static regions of the scene, since each pixel in a static region has the same expected value under random binary modulation. Hence, we propose coding only dynamic regions—the dynamic region-of-interest (RoI) can be determined by masking pixels whose coded exposures deviate significantly from one-another. As seen in Suppl. Fig. 2, the dynamic content may just be 25% of the image area, which provides significant scope for bandwidth savings. We observe that we can code just 25% of the total pixels, among additional compressive measurements, without a perceptible drop in visual quality, which yields an overall bandwidth requirement of $1.45\times$, or under twice the bandwidth cost of a single compressive measurement.

Results on More Sequences

Additional results are shown in Suppl. Fig. 3.



Supplementary Figure 2: **Coding dynamic regions can reduce readout of multi-bucket captures.** *(left column)* Dynamic regions are detected by computing the standard deviation of the coded exposures and thresholding them appropriately (e.g., by the 75^{th} percentile)—we show the dynamic regions in white here. Coded exposures are transmitted only in the dynamic regions. For the static regions, we simply use a long exposure—by adding multi-bucket captures. *(right column)* We observe that readout-bandwidth can be reduced to $1.75\times$ from $4\times$ in the case of a four-bucket capture with no perceptual degradation of reconstruction quality. Bandwidth is provided here as a multiple of the readout of a single compressive capture.



Supplementary Figure 3: **Results on additional sequences.** We use Hertz (Hz) to indicate the rate of emulation and frames-per-second (FPS) to indicate the frame-rate of the reconstructed video. Frame numbers are indicated in yellow font.

Supplementary Note 2. Event Cameras

Event-Generation Pseudocode

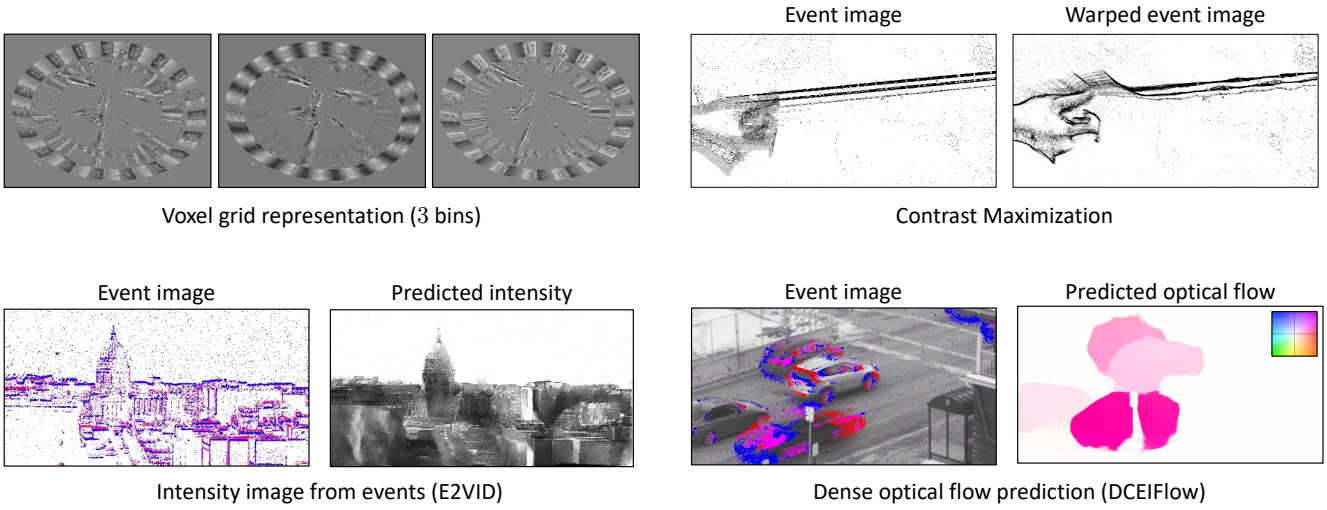
We provide the pseudocode for emulating events from photon-cubes in Algorithm 2. The contrast threshold τ and exponential smoothing factor β are the two parameters that determine the characteristics of the resulting event stream, such as its event rate (number of events per second). We use an initial time-interval T_0 (typically 80–100 bit-planes) to initialize the reference moving average, with T_0 being much smaller than T. The result of this pseudocode is an event-cube, $E_t(\mathbf{x})$, which is a sparse spatio-temporal grid of event polarities—positive spikes are denoted by 1 and negative spikes by -1. From the emulated event-cube, other event representations can be computed such as: an event stream, $\{(\mathbf{x}, t, p)\}$, where $p \in \{-1, 1\}$ indicates the polarity of the event; a frame of accumulated events [12] (seen in Figs. 4 and 9); and a voxel grid representation [27], where events are binned into a few temporal bins (shown in Suppl. Fig. 4 *(top left)* using 3 temporal bins).

Algorithm 2 Event Camera Emulation

Require: Photon-cube $B_t(\mathbf{x})$
Contrast threshold τ
Exponential smoothing factor, β
Pixel locations $\mathcal{X}$
Initial time-interval T_0, for computing reference moving average
Total bit-planes T
Ensure: Event-cube $E_t(\mathbf{x})$ that describes the spatio-temporal spikes
function EVENTCAMERAEMULATION($B_t(\mathbf{x})$, τ, β, T_0)
$E_t(\mathbf{x}) \leftarrow 0, \forall\, t, \forall\, \mathbf{x}$
for $\mathbf{x} \in \mathcal{X}$ **do**
Reference moving average, $\mu_{\text{ref}}(\mathbf{x}) \leftarrow 0$
Current moving average, $\mu_0(\mathbf{x}) \leftarrow 0$
for $1 \leq t \leq T_0$ **do**
$\mu_{\text{ref}}(\mathbf{x}) \leftarrow \beta\mu_{\text{ref}}(\mathbf{x}) + (1-\beta)B_t(\mathbf{x})$
end for
for $T_0 \leq t \leq T$ **do**
$\mu_t(\mathbf{x}) \leftarrow \beta\mu_{t-1}(\mathbf{x}) + (1-\beta)B_t(\mathbf{x})$
if $|\mu_t(\mathbf{x}) - \mu_{\text{ref}}(\mathbf{x})| > \tau$ **then**
$E_t(\mathbf{x}) \leftarrow \text{sign}(\mu_t(\mathbf{x}) - \mu_{\text{ref}}(\mathbf{x}))$
$\mu_{\text{ref}}(\mathbf{x}) \leftarrow \mu_{\text{ref}}(\mathbf{x}) + \tau * \text{sign}(\mu_t(\mathbf{x}) - \mu_{\text{ref}}(\mathbf{x}))$
end if
end for
end for
return $E_t(\mathbf{x})$
end function

Compatibility of SPAD-Events with Existing Event-Vision Algorithms

We now provide examples of downstream algorithms applied to SPAD-events, which shows the compatibility of the emulated event streams with existing event-vision algorithms. Supplementary Figure 4 shows three downstream algorithms with SPAD-events as their input: Contrast Maximization [8] which generates a warped image of events that has sharp edges (*top right*), E2VID [13] which estimates intensity frames from an event stream (*bottom left*), and DCEIFlow [21] which computes dense optical flow using intensity frames and aligned events (*bottom right*). Both E2VID and DCEIFlow use a voxel grid representation of events as their inputs. We include the visualization of a voxel grid representation in Suppl. Fig. 4 *(top left)*. All event streams were emulated using 3000 bit-planes of photon-cubes acquired at 96.8 kHz, and using $\beta = 0.95$ and $\tau = 0.4$ as emulation parameters. We note that the performance of these algorithms can be improved by finetuning pre-trained learning-based models on a dataset of SPAD-events.



Supplementary Figure 4: **Compatibility of SPAD-events with existing event-vision algorithms.** The flow field visualization follows Baker et al. [2]. The photon-cube for the contrast maximization output was obtained from Ma et al. [11].

Ablation of Brightness-Encoding Functions

Our event emulation scheme (Algorithm 2) relies on the SPAD's response curve to encode scene brightness, which is a non-linear and non-saturating response of the form

$$1 - \exp(-\alpha\Phi(\mathbf{x}, t)),$$

where $\alpha = \eta t_{\text{exp}}$ and assuming negligible dark count rate (DCR). Current event-cameras typically use a logarithmic response to encode scene brightness. This can also be utilized to emulate events from photon-cubes by setting h (as described in Eq. (8)) to be the log-MLE function:

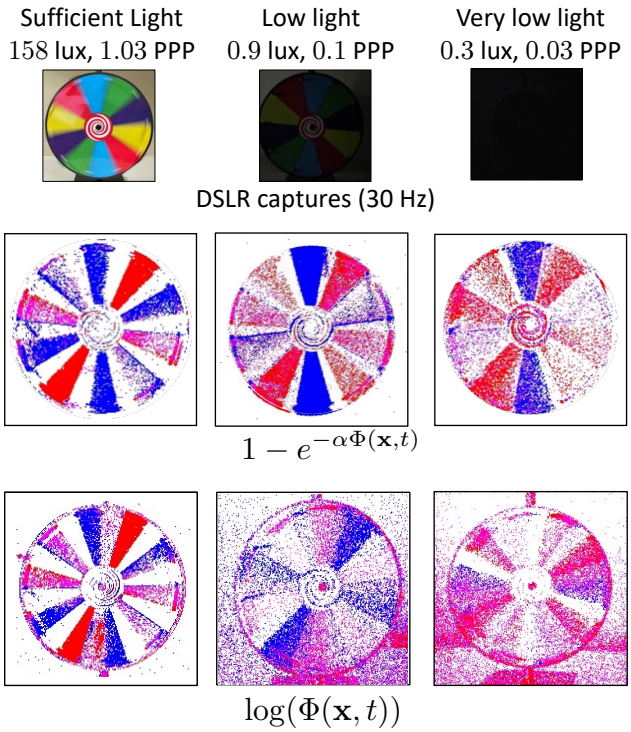
$$h(\mu) = \log\left(-\frac{\log(1-\mu)}{\eta t_{\text{exp}}}\right).$$

However, a log-response suffers from underflow issues, particularly at low-light scenarios as seen in Suppl. Fig. 5.

SoDaCam Flexibility and SPAD-Events

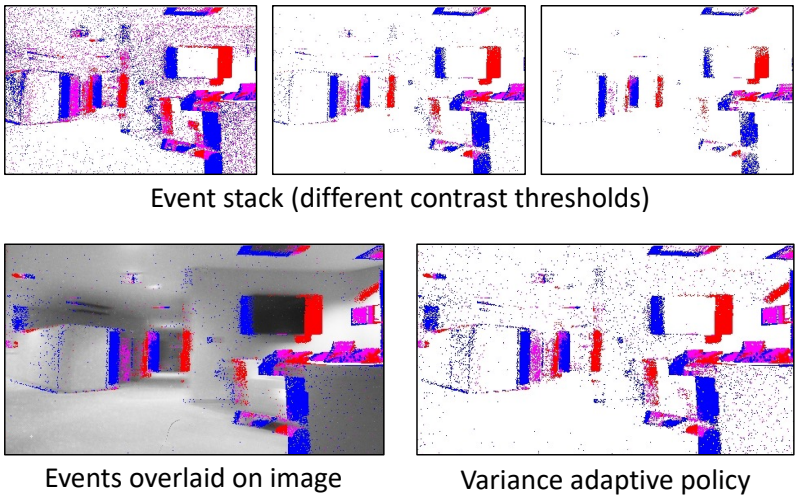
Here are a few benefits of the SoDaCam approach for event-based imaging:

- **Direct access to intensity information.** By computing a sum-image, SoDaCam makes intensity frames that are spatially- and temporally-aligned with the generated event stream available. This precludes the need for multiple devices, which often requires careful alignment and calibration.
- Further, the intensity frames obtained via the sum-image feature the SPAD's imaging capabilities, i.e., such intensity frames feature a high dynamic range and can be utilized in low-light imaging scenarios. This is in contrast to dynamic active vision sensors (DAVIS) [3, 6], where a conventional frame, which has limited dynamic range and low-light capabilities compared to SPAD-derived images, can be obtained in addition to the event stream.
- **Computing multiple event-streams simultaneously.** Contrast threshold τ is an important parameter that controls the sparsity and noise level of generated event stream: small values of τ can produce potentially noisy event streams that require extensive processing, while large values of τ can result in very sparse streams with less useful information. With SoDaCam, it is possible to emulate event streams with different values of τ simultaneously, thereby amortizing these trade-offs. In fact, this can be thought of analogous to exposure stacks but in the context of event-imaging. Supplementary Figure 6 *(top row)* shows an example of an 'event-image stack'.



Supplementary Figure 5: **Comparison of brightness encoding functions.** While the log-MLE is comparable to using the SPAD's response curve at ambient light levels, at low flux levels the underflow issues associated with the log function occur. Here, α denotes a sensor-determined and flux-independent constant.

- **Per-pixel contrast thresholds.** We can also vary the contrast threshold τ as a function of pixel location or incident intensity. For instance, we can use a smaller contrast threshold if we have an estimate of the incident intensity with less variance and a higher contrast threshold when there is more variance. We show an example of this in Suppl. Fig. 4 *(bottom right)*, where we vary the contrast threshold between 0.35 and 0.45 as a linear function of the sample variance of the moving average, $\mu_t(\mathbf{x})$.



Supplementary Figure 6: **Flexible event-based imaging**. *(top)* An 'event-stack' that employs increasing contrast thresholds, $\tau = 0.35, 0.4, 0.45$. *(bottom)* We can also output frames with aligned events, and use event generation policies that may not be trivial to realize in hardware—such as varying the contrast threshold τ as a linear function of the sample variance.

Supplementary Note 3. Motion Projections

Pseudocode for Emulating Motion Cameras

Algorithm 3 provides the pseudocode for emulating sensor motion from a photon-cube, where the sensor's trajectory is determined by the discretized function $\mathbf{r}$. At each time instant t, we shift bit-planes by $\mathbf{r}(t)$ and accumulate them in $\mathcal{I}_{\text{shift}}$. For pixels that are out-of-bounds, no accumulation is performed. For this reason, the number of summations that occur vary spatially across pixel locations $\mathbf{x}$—we normalize the emulated shift-image by the number of pixel-wise accumulations $N(\mathbf{x})$ to account for this. The function $\mathbf{r}$ can be obtained by discretizing any smooth 2D trajectory: by either rounding up or dithering, or by using a discrete line-drawing algorithm [4].

Algorithm 3 Motion Camera Emulation

Require: Photon-cube $B_t(\mathbf{x})$
  Discretized trajectory $\mathbf{r}(t)$
  Pixel locations $\mathcal{X}$
  Total bit-planes T
Ensure: $\mathcal{I}_{\text{shift}}(\mathbf{x})$
  function MOTIONCAMERAEMULATION($B_t(\mathbf{x})$, $\mathbf{r}$)
    $\mathcal{I}_{\text{shift}}(\mathbf{x}) \leftarrow 0, \forall\, \mathbf{x}$
    for $\mathbf{x} \in \mathcal{X}$ **do**
      Normalizer, $N(\mathbf{x}) \leftarrow 0$
      for $1 \leq t \leq T$ **do**
        if $\mathbf{x} + \mathbf{r}(t) \in \mathcal{X}$ **then**
          $N(\mathbf{x}) \leftarrow N(\mathbf{x}) + 1$
          $\mathcal{I}_{\text{shift}}(\mathbf{x}) \leftarrow \mathcal{I}_{\text{shift}}(\mathbf{x}) + B_t(\mathbf{x} + \mathbf{r}(t))$
        end if
      end for
      if $N(\mathbf{x}) > 0$ **then**
        $\mathcal{I}_{\text{shift}}(\mathbf{x}) \leftarrow \mathcal{I}_{\text{shift}}(\mathbf{x}) / N(\mathbf{x})$
      end if
    end for
  return $\mathcal{I}_{\text{shift}}(\mathbf{x})$
  end function

As described in Sec. 5.3, we consider two trajectories: linear and parabolic. Linear trajectories are parameterized by their slope

$$\mathbf{r}(t) = v\left(t - \frac{T}{2}\right)\hat{\mathbf{p}},$$

where v is the object velocity, $\hat{\mathbf{p}}$ is a unit vector that describes the trajectory's direction, and T is the total number of bit-planes. Parabolic trajectories are parameterized by their maximum absolute slope, v_{max}

$$\mathbf{r}(t) = \frac{v_{\text{max}}}{T}\left(t - \frac{T}{2}\right)^2 \hat{\mathbf{p}}.$$

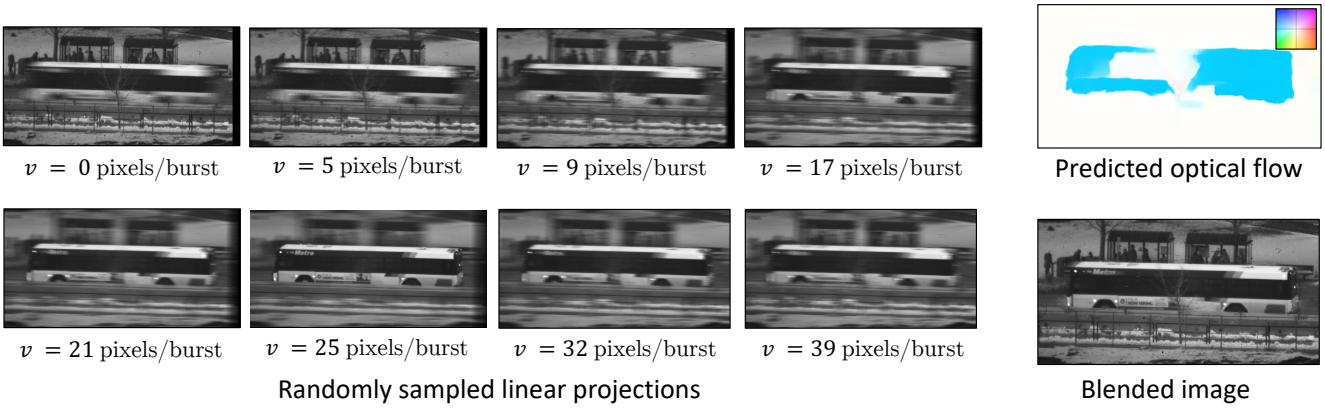
To prevent tail-clipping, which are image artifacts introduced by the finite extent of the parabolic integration, it is important to choose v_{max} to be sufficiently higher than the velocity of objects in the scene. Both linear and parabolic trajectories have a zero at $t = T/2$—which allows blending multiple linear projections without any pixel alignment issues.

Blending Multiple Linear Projections

As shown in Fig. 8, randomly sampling multiple linear projections (seen in Suppl. Fig. 7 *(left column)*) can provide motion compensation when only the motion direction, and not the exact extent of motion, is known. To blend these projections, in addition to the randomly sampled linear projections, we also compute two short exposures using bit-planes at the beginning and end of the photon-cube. For the scenes shown in Fig. 8, we used the first 200 and the last 200 bit-planes to emulate short exposures. We then use RAFT [17] to predict optical flow between the two short exposures—which can be used to select the

linear projection that can best compensate motion as a function of the pixel location (as seen in Suppl. Fig. 7 *(left column)*). We did not have to perform any spatial smoothing after selecting linear projections, since the optical flow predicted by RAFT was reasonably smooth.

Blending can also be achieved by choosing the least blurred linear projection in a per-pixel manner—similar to how focal stacking is typically achieved. This would however require predicting per-pixel blur kernels or constructing a measure of motion blur. Laplacian filters, which are typically used for focal stacking, do not readily work with motion stacks.



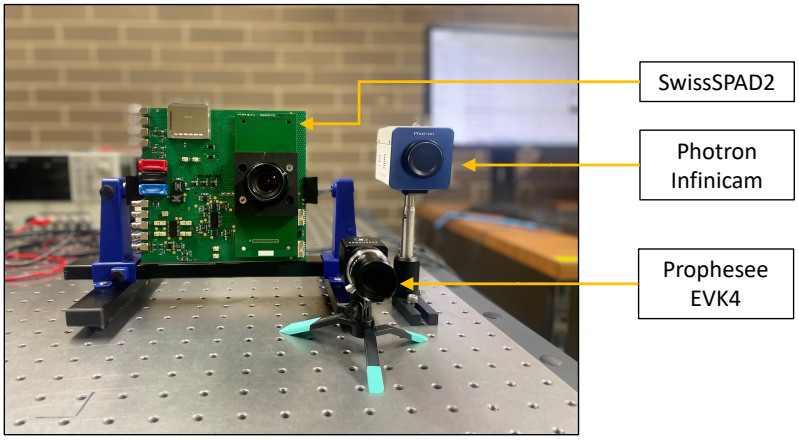
Supplementary Figure 7: **An example of motion stack blending.** *(left)* We sample 8 linear projections randomly along the road's orientation such that their ensuing pixel displacements are uniformly between 0 and 40 pixels. *(right)* We then blend them using the optical flow field that is predicted between two short exposures computed from the same photon-cube. The flow field visualization follows Baker et al. [2].

Supplementary Note 4. Experimental Setup for Secs. 6.1 and 6.2

Cameras and Sensory Arrays Used

We used the following imagers for our experiments described in Secs. 6.1 and 6.2:

- **SwissSPAD2 array** a 512×512 SPAD array that can be operated at a maximum frame-rate of 97 kHz. We operate the SwissSPAD2 in its 'half-array' mode: utilizing one of two sub-arrays, with a resolution of 512×256 pixels. The SPAD pixels have a pixel pitch of 16.4 μm and a low fill-factor of 10%, owing to the lack of microlenses in the prototype.
- **Prophesee EVK4 event camera**, which is a state-of-the-camera commercial event camera featuring a sensor resolution of 1280×720 pixels, pixel pitch of 4.86 μm and a fill-factor of $> 77\%$.
- **Photron infinicam** a conventional high-speed camera that can stream acquisition over USB-C at a resolution of 1246×1024 pixels and 1 kHz frame-rate. For higher frame-rates, it is necessary to reduce the number of rows that are read out—for the example, we use a resolution of 1246×240 pixels to obtain acquisition at 4 kHz in Fig. 10.



Supplementary Figure 8: **Cameras and sensor arrays used for the experiments described in Secs. 6.1 and 6.2.** The Infinicam and Prophesee were used for the comparisons made in Figs. 9 and 10 respectively.

Removing Hot Pixels

A few SPAD pixels (around 5% of the total pixels in our prototype) have extremely high dark current rate and therefore have $B_t(\mathbf{x}) = 1$ almost always. We detect these *hot pixels* by capturing a photon-cube of 100000 bit-planes in a very dark environment and detecting pixel locations with high photon counts. For video compressive and event imagers, we inpaint projections using OpenCV's implementation of the Telea algorithm [18]. For motion projections, we do not sum over bit-plane locations that correspond to hot pixels during integration. Further, we remove pixel locations from the hot pixel mask if the motion trajectory provides access to neighboring values that are not hot pixels. We inpaint the motion projection after excluding these points.

Experiment-wise Lens Specifications

We used C-mount lenses for our experiments with the following focal lengths:

- 12 mm for the comparison to Prophesee EVK4 in Fig. 9. The Prophesee EVK4 and the SwissSPAD2 were used with the same lens specifications.
- 16 mm for the coded exposures shown in Fig. 2.
- 35 mm for the spinning casino roulette shown in Figs. 4 and 10.
- 50 mm for the motion stack shown in Fig. 6 and the traffic scene shown in Fig. 8.
- 75 mm for the falling die sequence shown in Fig. 1, the measure tape sequence shown in Fig. 5, and the water splash captured in Fig. 7.

Supplementary Note 5. UltraPhase Experiments

Processor Description

The chip consists of a 3×6 array of processing cores, each of which can interface with 4×4 SPAD pixels via 3D stacking. At this point, the 3D stacking has not been completed, so we interface UltraPhase with the photon-cubes acquired by the SwissSPAD2 [19] instead. Every core is independent, has 4 kb of available RAM, and can execute programs of up to 256 instructions in length at a rate of 140 million instructions per second (MIPS). The system supports a wide range of instructions including, bit-wise operations, 32-bit arithmetic operations, data manipulation and custom inter-core synchronization. For more details, please refer to Ardelean [1].

We implement projections on UltraPhase by using a custom assembly code to program each core separately. We include the commented assembly code for all three projections in Listings 1 to 3. To compute multiple projections, we simply run projections sequentially, one bit-plane at a time. Since each projection can be computed significantly faster than the camera frame-rate (e.g., 1.678 ms for video compressive sensing of 40 Hz readout), this does not bottleneck acquisition. We include the processing time for each projection in Tab. 1.

Measuring Bandwidth

We assume that the outputs for sum, video compressive and motion projections have 12-bit depth. For event cameras, we assume that each event consists of 18-bits—-9-bits to encode the pixel location ($\lceil \log_2(12 \times 24) \rceil$), 8 bits to represent the timestamp (corresponding to the bit-plane index where the event was triggered), and 1-bit to encode polarity. We then measure readout on a 12×24 region-of-interest (RoI) of the falling die sequence that was acquired using the SwissSPAD2. Table 1 lists the readout bandwidth for each projection.

Measuring Power

The power consumption of UltraPhase is comprised of compute power and readout power. For compute power, the chip was characterized by executing instructions corresponding to each projection in an infinite loop and measuring its average power consumption. As an upper bound, we assumed the maximum possible power consumption for operations that involved reading and writing to the RAM. This measured power consumption was then scaled by the duty cycle of each projection—which is the ratio of the time required to process a bit-plane to the exposure time of each bit-plane.

For readout power, we consider a conventional digital interface at 3.3 V with a load of 7 pF operating at the specified bandwidth, amounting to 54 nanowatts for each kilobit readout (nW/kbps)—this is similar, for instance, to the USB interface utilized by the SwissSPAD2.

Table 1 provides the processing power, readout power and the total power for each projection. Clearly, processing requires an order of magnitude (or more) lesser power than readout, which explains how computing photon-cube projections results in reduced sensor power consumption.

Table 1: **Power and bandwidth benchmarks** when computing photon-cube projections on UltraPhase, a 24×12 array, at 40 Hz readout. We compare computing projections to reading out the entire photon-cube. We report the processing time, the readout bandwidth, and the compute and readout power for each projection.

	Processing time ↓	Bandwidth ↓	Power ↓		
	(ms)	(kbps)	Processing (μW)	Readout (μW)	Total (μW)
12-bit sum image	0.981	135	0.3	7.29	7.6
Snapshot compressive	1.678	135	3.0	7.29	10.3
Motion projection	1.096	135	1.3	7.29	8.6
Event camera	9.817	101.25	2.4	5.83	8.2
Three projections	12.591	405	6.7	21.87	28.6
Photon-cube readout	0.007	28125	5.4×10^{-3}	1518.8	1518.8

Table 2: **Power consumption** of SoDaCam versus conventional cameras (in mW), estimated for 512×256 pixels at 40 Hz readout. CMOS estimates assume the usage of column-parallel ADCs [15].

	Photon detection		Compute	Readout	Total	
	Dark	Ambient			Dark	Ambient
Photon-cube readout	1	62	-	690	691	752
Sum-image	1	62	0.3	4.5	5.8	66.8
VCS	1	62	1.3	4.5	6.8	67.8
Motion proj.	1	62	0.7	4.5	6.2	67.2
Event camera	1	62	1	3.6	5.6	66.6
Three proj.(s)	1	62	3	13.5	17.5	78.5
CMOS @ 40 FPS	~10–25		-	4.5	~15–30	
CMOS @ 4k FPS	~600–2500		-	450	~1000–3000	

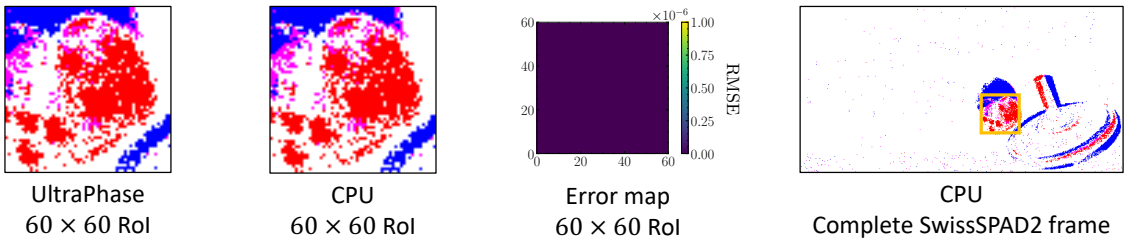
Comparison to CMOS Sensors

In addition to compute and readout power quantified in the previous section, a computational SPAD also consumes power to detect photons. By incorporating this photon-detection power, we can provide a rough comparison of SoDaCam projections to CMOS sensors. The photon-detection dissipation depends on the number of photon detections, and hence varies with the light level—for the SwissSPAD2, this is measured to be $<$ 1mW in the dark and ~62 mW in indoor lighting [19]. To estimate compute and readout power, we linearly scale the measurements presented in Tab. 1 for an array of 512×256 pixels. We note that this is a conservative estimate, since UltraPhase is not designed to be a low-power device.

As seen in Tab. 1, under ambient lighting, the power consumption of our emulated cameras is higher than conventional CMOS cameras; while in low-light, the SPAD consumes lesser power owing to fewer photon detections. We remark that without the bandwidth reduction facilitated by photon-cube projections, SPADs are at a considerable disadvantage compared to their CMOS counterparts. Finally, we provide a comparison against high-speed CMOS cameras, which can also be used to obtain photon-cube projections, albeit with a read noise penalty (Sec. 6.2), and higher power consumption.

Visualization of Projections

Since UltraPhase is a low-resolution sensor-processor (12×24 pixels), we visualize projections by repeating computations in a tiled manner to cover a region-of-interest (RoI) of 60×60 pixels. Supplementary Figure 9 shows the visualization of an event camera emulated on UltraPhase. To provide more context, we include the CPU visualization of the entire SwissSPAD2 event-frame. We also verified that the outputs of UltraPhase were identical to CPU-run outputs by computing the RMSE between event frames that result from UltraPhase computations and CPU computations (see error map in Suppl. Fig. 9).



Supplementary Figure 9: **Event camera computed using UltraPhase,** on 2500 bit-planes of the falling die sequence. For visualization purposes, we run computations on UltraPhase in a tiled manner so as to cover a RoI of 60×60 pixels. We compare this to CPU-run outputs of the same RoI and verify that they are identical. For context, we highlight this RoI using a bounding box on the CPU-run event-frame that has a resolution of 256×512 pixels. The event simulation parameters used were $\tau = 0.45, \beta = 0.95, T_0 = 80$.

Listing 1: **Custom assembly code for implementing video compressive sensing on UltraPhase.** Here, we consider computing one compressive snapshot that is multiplexed by 16 binary random masks.

```
--RAM(64..127) stores the 64 subframe compression code masks as one bit per pixel in byte1 and byte0
--The output is available in RAM(0..15)
--CtrlOut is strobed after every binary frame

#define pixelValue 0                                    --register R0 is used for the pixel values
#define frameIdx 1                                          --register R1 is used as a counter for the
↪  current binary frame
#define subFrameIdx 2                                  --register R2 is used as a counter for the current
↪  subFrame
#define mask 3                                              --register R3 is used to store the
↪  compression codes for the current subframe
#define aux0 4                                              --register R4 is used for
#define aux1 5                                              --register R5 is used for
#define CtrlOut 0b10000                                  --address for external trigger signal
#define toMask 0b001000                                  --address used to fetch to R3 (mask)

0: LOAD subFrameIdx, $64, 0                           --initialize subFrame pointer to 0 + 64 (RAM offset)
1: LOAD subFrameIdx, $0, 1
2: LOAD frameIdx, $16, 0                          --initialize frameIdx pointer to 16 i.e. number of
↪  bit_planes_per_subframe
3: LOAD frameIdx, $0, 1

4: GETP 5, 1, 0                                       --get pixel data and store it in R0
↪  (pixelValue)
5: FETCH @subFrameIdx, toMask, 0        --get the appropriate mask
6: AND pixelValue, mask, pixelValue        --apply mask i.e. multiply with code[subframe]
7: CALL 50                                                    --accumulate pixels
8: TELL CtrlOut, 0                                      --strobe CtrlOut

-------------------------------------------------
-- update indexes
9: OR aux0, aux0, aux0                                --clear flags
10: SUBC frameIdx, $1                                 --decrement frameIdx
11: JUMPZ 13                                               --check if we finished with all the binary
↪  planes per subframe
12: JUMP 4                                                       --if not, then move on
13:  OR aux0, aux0, aux0                          --clear flags
14: ADDC subFrameIdx, $1                          --move to next subFrame
15: JUMP 2                                                        --reset frameIdx and continue

-------------------------------------------------
-- this subroutine will accumulate the pixel values from pixelValue
50: LOAD aux0, $16, 0                                 --set R4 (aux0) to 16
51: LOAD aux0, $0, 1
52: LOAD aux1, $0, 0                                 --set R5 (aux1) to 0
53: LOAD aux1, $0, 1
54: OR aux0, aux0, aux0                                 --clear flags
55: SR0 pixelValue, pixelValue                --extract one pixel
56: ADDC (aux0), aux1, (aux0)                --increment if pixel is 1
57: SUBC aux0, $1                                          --decrement counter
58: JUMPNZ 54                                                 --repeat until done with all 16 pixels
59: RET
```

Listing 2: **Custom assembly code for implementing event cameras on UltraPhase.**

```
--The core will strobe CtrlOut every time an evet took place and the SoC needs to read RAM(0) to get it

#define toaux0 0b000010
#define toaux1 0b000100
#define pixelValue 0        --register R0 is used for the binary pixel values
#define aux0 1                      --register R1 is used for misc
#define aux1 2                      --register R2 is used for misc
#define Bpointer 3               --register R3 is used as a pointer to the reference_average stored in
↪  RAM(17..32)
#define Apointer 4               --register R4 is used as a pointer to the current_average stored in
↪  RAM(1..16)
#define pixelIdx 5               --register R5 is used for the current pixel index
#define CtrlOut 0b10000                           --address for external trigger signal
#define NOut        0b01000                          --address for north trigger signal
#define decayAddress 127                      --RAM address 127 stores the exponential_decay
#define decayComplementAddress 126        --RAM address 126 stores the value for 1-exponential_decay
#define intervalAddress 125                     --RAM address 125 stores the initial_interval
#define thresholdAddress 124               --RAM address 124 storesthe contrast_threshold
#define frameIdxAddress 123                     --RAM address 123 stores the frame_index counter


0: GETP 5, 1, 0                                                                  --get pixel data and
↪  store it in R0 (pixelValue)
1: LOAD pixelIdx, $0, 1                                                         --set R5 (pixelIdx) to
↪  16 to use as counter for the pixels
2: LOAD pixelIdx, $16, 0
3: LOAD Apointer, $0, 1                                                         --initialize pointers
4: LOAD Apointer, $1, 0
5: LOAD Bpointer, $0, 1
6: LOAD Bpointer, $17, 0

7: FETCH @decayAddress, toaux0, 0                                --get decay constant from RAM and
↪  store it into R1 (aux0)
8: FETCH @decayComplementAddress, toaux1, 0               --get (1 - decay) constant from RAM and
↪  store it into R2 (aux1)

9: MUL (Apointer), aux0, (Apointer)                                -- current_average[pixelIdx] -
↪  exponential_decay * current_average[pixelIdx]
10: SRX (Apointer), (Apointer)                                        -- 8 fractional bit
↪  multiplication
11: SRX (Apointer), (Apointer)
12: SRX (Apointer), (Apointer)
13: SRX (Apointer), (Apointer)
14: SRX (Apointer), (Apointer)
15: SRX (Apointer), (Apointer)
16: SRX (Apointer), (Apointer)
17: SRX (Apointer), (Apointer)

18: SR0 pixelValue, pixelValue                                          --shift R0 (pixelValue) to the
↪  right and pad with 0; the pixel bit is loaded into the carry flag
19: JUMPNC 21
↪                                                                                --current_average[pixelIdx]
↪  - current_average[pixelIdx] + (1-exponential_decay) * 0
20: ADD (Apointer), aux1, (Apointer)                          --current_average[pixelIdx] -
↪  current_average[pixelIdx] + (1-exponential_decay) * 1
21: FETCH @frameIdxAddress, toaux0, 0                  --get frame_index from RAM and store it into
↪  R1 (aux0)
22: FETCH @intervalAddress, toaux1, 0                  --get initial_interval from RAM and store it
↪  into R2 (aux1)
23: CMP aux0, aux1                                                              --check if
↪  frame_index < initial_interval
24: JUMPNC 40                                                                      --if not, jump and
↪  process
```

```
25: FETCH (Apointer), toaux0, 0                                    --if yes, get
↪  current_average[pixelIdx]
26: STORE aux0, (Bpointer)                                          --reference_average[pixelIdx]
↪  = current_average[pixelIdx]
27: JUMP 70                                                                    --GO TO NEXT
↪  PIXEL

-- frame_index is larger than initial_interval
40: FETCH (Apointer), toaux0, 0                                --if not, get
↪  current_average[pixelIdx] and store it into R1 (aux0)
41: FETCH (Bpointer), toaux1, 0                                    --get
↪  reference_average[pixelIdx] and store it into R2 (aux1)
42: SUB aux0, aux1, aux0                                            --diff[pixelIdx] =
↪  current_average[pixelIdx] - reference_average[pixelIdx] and store it into R1 (aux0)
43: JUMPNC        60                                                            --diff is
↪  positive

-- compare abs(diff) with threshold if diff is negative
44: FETCH @thresholdAddress, toaux1, 0                  --get contrast_threshold and store it
↪  into R2 (aux1)
45: NEG aux1, aux1                                                      --diff is negative,
↪  so make contrast_threshold negative and store it into R2 (aux1)
46: CMP aux0, aux1                                                      --if diff <
↪  -contrast_threshold
47: JUMPNC 70                                                             --if not, GO TO
↪  NEXT PIXEL
48: NEG pixelIdx, @0                                                --RAM(0) = -pixel_index
↪  i.e. a negative event
49: ADD (Bpointer), aux1, (Bpointer)                --reference_average += contrast_threshold *
↪  (-1)
50: TELL CtrlOut, 0                                                   --strobe CtrlOut to
↪  signal an event
51: JUMP 70                                                                    --GO TO NEXT
↪  PIXEL

-- compare abs(diff) with threshold if diff is positive
60:  FETCH @thresholdAddress, toaux1, 0                  --get contrast_threshold and store it
↪  into R2 (aux1)
61: CMP aux1, aux0                                                      --if diff >
↪  contrast_threshold
62: JUMPNC 70                                                             --if not, GO TO
↪  NEXT PIXEL
63: STORE pixelIdx, @0                                              --RAM(0) = pixel_index
↪  i.e. a positive event
64: ADD (Bpointer), aux1, (Bpointer)                --reference_average += contrast_threshold *
↪  1
65: TELL CtrlOut, 0                                                   --strobe CtrlOut to
↪  signal an event
66: JUMP 70                                                                    --GO TO NEXT
↪  PIXEL

--GO TO NEXT PIXEL
70: OR pixelValue, pixelValue, pixelValue                --clear flags
71: ADDC Apointer, $1                                                --increment Apointer
72: ADDC Bpointer, $1                                                --increment Bpointer
73: SUBC pixelIdx, $1                                                --decrement pixel counter
74: JUMPNZ 7                                                                --if not done with
↪  pixels, go to next one
75: FETCH @frameIdxAddress, toaux0, 0                 --if done with pixels, increment frame
↪  counter and get new pixel values
76: TELL NOut, 0                                                      --strobe NOut to signal
↪  a new exposure
77: ADDC aux0, $1
78: STORE aux0, @frameIdxAddress
79: JUMP 0
```

Listing 3: **Custom assembly code for implementing motion projections on UltraPhase.** Without loss of generality, we consider a linear projection along the horizontal direction.

```
--The projection is available in RAM(0..15)
-- CtrlOut is strobed after every frame

#define Xshift 0                --register R0 is used for the horizontal shift
#define timestep 1                --register R1 is used for the current timestep
#define origPixels 2         --register R2 is used for the current core's pixels
#define shiftPixels 3         --register R3 is used for the shifted pixels
#define aux0 4                          --register R4 is used for misc
#define aux1 5                          --register R5 is used for misc
#define CtrlOut 0b10000                      --address for external trigger signal
#define shiftL3neighAddr 127         --RAM(127) stores the 0b0111_0111_0111_0111 mask
#define shiftL3currAddr 126                  --RAM(126) stores the 0b1000_1000_1000_1000 mask
#define shiftL2neighAddr 125         --RAM(125) stores the 0b0011_0011_0011_0011 mask
#define shiftL2currAddr 124                  --RAM(124) stores the 0b1100_1100_1100_1100 mask
#define shiftL1neighAddr 123         --RAM(123) stores the 0b0001_0001_0001_0001 mask
#define shiftL1currAddr 122                  --RAM(122) stores the 0b1110_1110_1110_1110 mask

0: LOAD Xshift, $0xFFF8, 0        --load -8 into Xshift as initial value
1: LOAD Xshift, $0xFFFF, 1
2: CALL 50                                      --get the correct pixel values according to Xshift
3: CALL 30                                      --accumulate pixels
4: CALL 20                                      --advance time
5: TELL CtrlOut, 0                        -- strobe CtrlOut to signal a new frame
6: JUMP 2                                   --repeat

-----------------------------------------------------
-- this subroutine will advance timestep and update Xshift
20: OR timestep, timestep, timestep         --clear flags
21: ADDC timestep, $1                                --increment timestep
22: OR timestep, timestep, aux0                --copy timestep to aux0
23: SPLIT aux0, 5, 0b010000                        --timestep = timestep/1024
24: SR0 aux0, aux0
25: SR0 aux0, aux0
26: LOAD aux1, $1, 0                                --load aux1 with the Xspeed value of 1
27: MAC aux0, aux1, Xshift, Xshift         --Xshift += Xspeed*timestep/2
28: RET

-----------------------------------------------------
-- this subroutine will accumulate the pixel values from shiftPixels
30: LOAD aux0, $16, 0                                --set R4 (aux0) to 16
31: LOAD aux0, $0, 1
32: LOAD aux1, $0, 0                                --set R5 (aux1) to 0
33: LOAD aux1, $0, 1
34: OR aux0, aux0, aux0                                --clear flags
35: SR0 shiftPixels, shiftPixels         --extract one pixel
36: ADDC (aux0), aux1, (aux0)                --increment if pixel is 1
37: SUBC aux0, $1                                --decrement counter
38: JUMPNZ 34                                        --repeat until done with all 16 pixels
39: RET

-----------------------------------------------------
-- this subroutine will get pixel values from the core and the correct neighbour based on the Xshift
50: GETP 5, 4, 0                                --get pixel values and store them in R2
↪  (origPixels)
51: OR Xshift, Xshift, aux1                        --copy current shift into R5 (aux1)
52: LOAD aux0, $0, 0                                --set R4 (aux0) to 0 to use for Xshift comparison
53: LOAD aux0, $0, 1
54: CMP aux1, aux0                                        --compare current shift with 0
55: JUMPZ 115                                        --current shift = 0
56: JUMPC 120                                        --current shift < 0
57: JUMP 58                                                --current shift > 0
-------------------------------
-- positive shifts
```

```
58: LOAD aux0, $4, 0                                --set R4 (aux0) to 4 for current shift comparison
59: LOAD aux0, $0, 1
60: CMP aux0, aux1
61: JUMPC 105                       --current shift > 4, read from the neighbour's neighbour
62: JUMPZ 100                                         --current shift is 4
63: LOAD aux0, $2, 0                                --set R4 (aux0) to 2, for current shift comparison
64: CMP aux0, aux1
65: JUMPC 90                                          --current shift is 3
66: JUMPZ 80                                         --current shift is 2

-- shift is 1
67: SL0 shiftPixels, shiftPixels                     --pixels from the neighbour need to be shifted
↪  to the left 3 times
68: SL0 shiftPixels, shiftPixels
69: SL0 shiftPixels, shiftPixels
70: OR origPixels, origPixels, aux1                    --save curent pixels in the aux1 variable
71: SR0 aux1, aux1                                                --pixels from this core need
↪  to be shifted to the right once
72: AND shiftPixels, @shiftL3currAddr, shiftPixels      --apply mask to select relevant bits from
↪  neighbour
73: AND aux1, @shiftL3neighAddr, aux1                   --apply mask to select relevant bits from
↪  current core
74: OR shiftPixels, aux1, shiftPixels                   --combine to create final pixel values
75: RET

-- shift is 2
80: SL0 shiftPixels, shiftPixels                                  --pixels from neighbour need to
↪  be shifted to the left 2 times
81: SL0 shiftPixels, shiftPixels
82: OR origPixels, origPixels, aux1                            --save curent pixels in the aux1
↪  variable
83: SR0 aux1, aux1                                                                --pixels from
↪  this core need to be shifted to the right 2 times
84: SR0 aux1, aux1
85: AND shiftPixels, @shiftL2currAddr, shiftPixels      --apply mask to select relevant bits from
↪  neighbour
86: AND aux1, @shiftL2neighAddr, aux1                     --apply mask to select relevant
↪  bits from current core
87: OR shiftPixels, aux1, shiftPixels                     --combine to create final pixel
↪  values
88: RET

-- shift is 3
90: SL0 shiftPixels, shiftPixels                                  --pixels from neighbour need to
↪  be shifted to the left once
91: OR origPixels, origPixels, aux1                            --save curent pixels in the aux1
↪  variable
92: SR0 aux1, aux1                                                                --pixels from
↪  this core need to be shifted to the right 3 times
93: SR0 aux1, aux1
94: SR0 aux1, aux1
95: AND shiftPixels, @shiftL1currAddr, shiftPixels      --apply mask to select relevant bits from
↪  neighbour
96: AND aux1, @shiftL1neighAddr, aux1                     --apply mask to select relevant
↪  bits from current core
97: OR shiftPixels, aux1, shiftPixels                     --combine to create final pixel
↪  values
98: RET

-- shift is 4
100: PUTN origPixels, 4                        --shift your pixels to the left (share pixels with
↪  neighbour)
101: SAVEN 1                                      --save pixels from rigth neighbour
102: GETN 0, 8, 0                               --get pixels from rigth neighbour and store in
↪  shiftPixels
103: RET
```

```
-- shift is > 4
105: PUTN origPixels, 4                        --shift your pixels to the left (share pixels with
↪  neighbour)
106: SAVEN 1                                      --save pixels from right neighbour
107: GETN 0, 8, 0                               --get pixels from right neighbour and store in
↪  shiftPixels
108: OR aux1, aux1, aux1                  --clear Carry flag
109: SUBC aux1, $4                               --subtract 4 from current shift because we read from
↪  a neighbour
110: JUMP 58

------------------------------
-- shift is zero
115: OR origPixels, origPixels, shiftPixels  --the shift is zero, so keep the pixels
116: RET

------------------------------
-- negative shifts
120: LOAD aux0, $FFFC, 0                  --set R4 (aux0) to -4 to use for current shift comparison
121: LOAD aux0, $0xFFFF, 1
122: CMP aux1, aux0
123: JUMPC 170                                    --current shift < -4 so we need to read from
↪  neighbour's neighbour
124: JUMPZ 160                                    --current shift is -4
125: LOAD aux0, $FFFE, 0                  --set R4 (aux0) to -2 to use for current shift comparison
126: CMP aux1, aux0
127: JUMPC 150                                     --current shift is -3
128: JUMPZ 140                                    --current shift is -2

-- shift is -1
129: SR0 shiftPixels, shiftPixels                                                --pixels from
↪  neighbour need to be shifted to the right 3 times
130: SR0 shiftPixels, shiftPixels
131: SR0 shiftPixels, shiftPixels
132: OR origPixels, origPixels, aux1                                    --save curent pixels in
↪  the aux1 variable
133: SL0 aux1,
↪  aux1                                                                          --pixels from
↪  this core need to be shifted to the left once
134: AND shiftPixels, @shiftL1neighAddr, shiftPixels        --apply mask to select relevant bits from
↪  neighbour
135: AND aux1, @shiftL1currAddr, aux1                                   --apply mask to select
↪  relevant bits from current core
136: OR shiftPixels, aux1, shiftPixels                                   --combine to create final
↪  pixel values
137: RET

-- shift is -2
140: SR0 shiftPixels, shiftPixels                                              --pixels from
↪  neighbour need to be shifted to the right 2 times
141: SR0 shiftPixels, shiftPixels
142: OR origPixels, origPixels, aux1                                    --save curent pixels in
↪  the aux1 variable
143: SL0 aux1,
↪  aux1                                                                          --pixels from
↪  this core need to be shifted to the left 2 times
144: SL0 aux1, aux1
145: AND shiftPixels, @shiftL2neighAddr, shiftPixels        --apply mask to select relevant bits from
↪  neighbour
146: AND aux1, @shiftL2currAddr, aux1                                   --apply mask to select
↪  relevant bits from current core
147: OR shiftPixels, aux1, shiftPixels                                   --combine to create final
↪  pixel values
148: RET

-- shift is -3
```

```
150: SR0 shiftPixels, shiftPixels                                        --pixels from
↪  neighbour need to be shifted to the right once
151: OR origPixels, origPixels, aux1                                    --save curent pixels in
↪  the aux1 variable
152: SL0 aux1,
↪  aux1                                                                 --pixels from
↪  this core need to be shifted to the left 3 times
153: SL0 aux1, aux1
154: SL0 aux1, aux1
155: AND shiftPixels, @shiftL3neighAddr, shiftPixels        --apply mask to select relevant bits from
↪  neighbour
156: AND aux1, @shiftL3currAddr, aux1                                   --apply mask to select
↪  relevant bits from current core
157: OR shiftPixels, aux1, shiftPixels                                   --combine to create final
↪  pixel values
158: RET

-- shift is -4
160: PUTN origPixels, 1                        --shift your pixels to the right (share pixels with
↪  neighbour)
161: SAVEN 4                                      --save pixels from left neighbour
162: GETN 2, 8, 0                               --get pixels from left neighbour and store in
↪  shiftPixels
163: RET

-- shift is < -4
170: PUTN origPixels, 1                        --shift your pixels to the right (share pixels with
↪  neighbour)
171: SAVEN 4                                      --save pixels from left neighbour
172: GETN 2, 8, 0                               --get pixels from left neighbour and store in
↪  shiftPixels
173: OR aux1, aux1, aux1                 --clear Carry flag
174: ADDC aux1, $4                                --add 4 to current shift because we read from a
↪  neighbour
175: JUMP 120
```

Supplementary References

[1] A. Ardelean. *Computational Imaging SPAD Cameras*. PhD thesis, École polytechnique fédérale de Lausanne, 2023. 12

[2] S. Baker, D. Scharstein, J. Lewis, S. Roth, M. J. Black, and R. Szeliski. A database and evaluation methodology for optical flow. *International journal of computer vision*, 92:1–31, 2011. 6, 10

[3] C. Brandli, R. Berner, M. Yang, S.-C. Liu, and T. Delbruck. A 240 × 180 130 dB 3 µs latency global shutter spatiotemporal vision sensor. *IEEE Journal of Solid-State Circuits*, 49(10):2333–2341, 2014. doi: 10.1109/JSSC.2014.2342715. 6

[4] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems journal*, 4(1):25–30, 1965. 9

[5] S. H. Chan, X. Wang, and O. A. Elgendy. Plug-and-play admm for image restoration: Fixed-point convergence and applications. *IEEE Transactions on Computational Imaging*, 3(1):84–98, 2017. doi: 10.1109/TCI.2016.2629286. 2

[6] S. Chen and M. Guo. Live demonstration: Celex-v: A 1m pixel multi-mode event-based sensor. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2019. 6

[7] X. Y. Chengshuai Yang, Shiyu Zhang. Ensemble learning priors driven deep unfolding for scalable video snapshot compressive imaging. In *IEEE European Conference on Computer Vision (ECCV)*, 2022. 2

[8] G. Gallego, H. Rebecq, and D. Scaramuzza. A unifying contrast maximization framework for event cameras, with applications to motion, depth, and optical flow estimation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018. 5

[9] Y. Li, M. Qi, R. Gulve, M. Wei, R. Genov, K. N. Kutulakos, and W. Heidrich. End-to-end video compressive sensing using anderson-accelerated unrolled networks. In *2020 IEEE International Conference on Computational Photography (ICCP)*, pages 1–12, 2020. doi: 10.1109/ICCP48838.2020.9105237. 2

[10] Y. Liu, X. Yuan, J. Suo, D. J. Brady, and Q. Dai. Rank minimization for snapshot compressive imaging. *IEEE Trans. Pattern Anal. Mach. Intell.*, 41(12):2990 – 3006, 2019. doi: 10.1109/TPAMI.2018.2873587. URL https://doi.org/10.1109/TPAMI.2018.2873587. 2

[11] S. Ma, S. Gupta, A. C. Ulku, C. Bruschini, E. Charbon, and M. Gupta. Quanta burst photography. *ACM Transactions on Graphics*, 39(4):1–16, July 2020. ISSN 0730-0301, 1557-7368. 6

[12] A. I. Maqueda, A. Loquercio, G. Gallego, N. García, and D. Scaramuzza. Event-based vision meets deep learning on steering prediction for self-driving cars. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018. 5

[13] H. Rebecq, R. Ranftl, V. Koltun, and D. Scaramuzza. Events-to-video: Bringing modern computer vision to event cameras. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019. 5

[14] P. Shedligeri, A. S, and K. Mitra. A unified framework for compressive video recovery from coded exposure techniques. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, pages 1600–1609, January 2021. 2

[15] M. F. Snoeij, A. J. P. Theuwissen, K. A. A. Makinwa, and J. H. Huijsing. Multiple-Ramp Column-Parallel ADC Architectures for CMOS Image Sensors. *IEEE JSSC*, 2007. doi: 10.1109/JSSC.2007.908720. 13

[16] M. Tassano, J. Delon, and T. Veit. Fastdvdnet: Towards real-time deep video denoising without flow estimation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020. 2

[17] Z. Teed and J. Deng. Raft: Recurrent all-pairs field transforms for optical flow. In *European Conference on Computer Vision*, 2020. 9

[18] A. Telea. An image inpainting technique based on the fast marching method. *Journal of graphics tools*, 9(1):23–34, 2004. 11

[19] A. C. Ulku, C. Bruschini, I. M. Antolovic, Y. Kuo, R. Ankri, S. Weiss, X. Michalet, and E. Charbon. A 512 × 512 SPAD Image Sensor With Integrated Gating for Widefield FLIM. *IEEE Journal of Selected Topics in Quantum Electronics*, 25(1):1–12, Jan. 2019. ISSN 1077-260X, 1558-4542. doi: 10.1109/JSTQE.2018.2867439. 12, 13

[20] S. V. Venkatakrishnan, C. A. Bouman, and B. Wohlberg. Plug-and-play priors for model based reconstruction. In *2013 IEEE Global Conference on Signal and Information Processing*, pages 945–948, 2013. doi: 10.1109/GlobalSIP.2013.6737048. 2

[21] Z. Wan, Y. Dai, and Y. Mao. Learning dense and continuous optical flow from an event camera. *IEEE Transactions on Image Processing*, 31:7237–7251, 2022. doi: 10.1109/TIP.2022.3220938. 5

[22] Z. Wang, H. Zhang, Z. Cheng, B. Chen, and X. Yuan. Metasci: Scalable and adaptive reconstruction for video compressive sensing. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2083–2092, June 2021. 2

[23] Z. Wu, J. Zhang, and C. Mou. Dense deep unfolding network with 3D-CNN prior for snapshot compressive imaging. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 4892–4901, October 2021. 2

[24] X. Yuan. Generalized alternating projection based total variation minimization for compressive sensing. In *2016 IEEE International Conference on Image Processing (ICIP)*, pages 2539–2543, 2016. doi: 10.1109/ICIP.2016.7532817. 2

[25] X. Yuan, D. J. Brady, and A. K. Katsaggelos. Snapshot compressive imaging: Theory, algorithms, and applications. *IEEE Signal Processing Magazine*, 38(2):65–88, 2021. 2

[26] X. Yuan, Y. Liu, J. Suo, F. Durand, and Q. Dai. Plug-and-play algorithms for video snapshot compressive imaging. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(10):7093–7111, 2022. doi: 10.1109/TPAMI.2021.3099035. 2

[27] A. Z. Zhu, L. Yuan, K. Chaney, and K. Daniilidis. Unsupervised event-based learning of optical flow, depth, and egomotion. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 989–997, 2019. 5