# Appendices

## Table of Contents

In the following appendices, we provide more technical and experiment-specific details—and some interesting practical considerations and asides. We categorically list each title according to the section in the main text that they correspond to.

*Please note: citation numbers in the appendices point to the main paper's references.*

## Sec 2: Extended Related Works

We now provide an expanded discussion of works that are related to quanta neural networks either in terms of its technical components or pertinent techniques that have been applied to other sensing modalities beyond quanta image sensors.

**State-space models in computer vision.** Our work presents a connection between integrators that perform motion-aware aggregation of photon detections and state-space models, which forms the basis of our QNN layers. Recent state-space models [10, 23, 24, 64, 65] offer temporal modeling capabilities with a subquadratic (in sequence length) complexity. Several works examine their potential as a replacement for vision transformers [43, 46, 55]. State-space models have also been utilized for image restoration [25, 62, 79], object detection [12, 72], semantic segmentation [77], and even processing event-sensor outputs [29, 59, 87]. We refer readers to recent survey papers for a comprehensive list [57, 74].

**Processing high-speed spike camera outputs.** QNNs, with modifications to its handcrafted layer (or by simply learning the first layer, like we do for intermediate layers), may be applied to other temporally-oversampled sensors. An example of such a temporally-oversampled sensor is a spike camera [30, 84], which outputs binary-valued responses at around 40 kHz; the sensor continuously accumulates photons and fires a "1" when a certain threshold is exceeded (the accumulator is reset shortly after). There have been several works that reconstruct images from spike streams, by exploiting their statistical properties [81, 84], and optionally using learned neural network to improve output image fidelity [7, 78, 82, 85]. Quanta neural networks may be applied for image reconstruction from spike camera outputs—and more broadly for other computer vision tasks—by integrating temporal information over longer context windows. While we do not study QNN's applicability to spike streams in this work, such a study would be informative but subject to the public availability of suitable (spike camera) datasets.

## Sec 3.2: Adaptive-Exponential Smoothing

The goal of Bayesian run-length estimation, as employed in the Bayesian online change-point detection algorithm (BOCPD [1]), is to determine the time since the last change occurred. Changes can be detected in BOCPD by flagging time instances where the forecaster associated with run length zero has a greater value than the forecaster that predicts that the run length has been incremented from its previously known value. Sundar et al. [67] use BOCPD to set the window lengths of cumulative moving averages, *i.e.*, whether to continue updating the stored cumulative mean or to reset it when a change is detected. In contrast, our formulation of the adaptive integrator in Sec. 3.2 does not rely on change detection; we use run length estimation to modulate the adaptive exponential smoothing operator.

In the following, we provide more details regarding the forecasters $\{\nu_s(\mathbf{p})\}_{s=1}^{S}$, where $S$ is the (constant) number of forecasters maintained per pixel, and $\mathbf{p}$ denotes the pixel location, used in Bayesian run-length estimation. Following the main text, we now drop the pixel index to describe the initialization, pruning, and updates of the forecaster more concisely.

**Forecaster initialization and update rule.** Each forecaster $\nu_s$ is associated with a Beta prior, which is parameterized by two values $\alpha_s$ and $\beta_s$. At each time step, we initialize a new forecaster $\tilde{\nu}$ as:

$$\tilde{\nu} \leftarrow \gamma \sum_{s=1}^{S} \nu_s \left( \frac{\alpha_s}{\alpha_s + \beta_s} B_t + \frac{\beta_s}{\alpha_s + \beta_s} (1 - B_t) \right). \quad (10)$$

The newly initialized forecaster is associated with a uniform prior, *i.e.*, Beta($\tilde{\alpha} = 1, \tilde{\beta} = 1$). Next, we update existing forecasters

$$\nu_s \leftarrow (1 - \gamma)\nu_s \left( \frac{\alpha_s}{\alpha_s + \beta_s} B_t + \frac{\beta_s}{\alpha_s + \beta_s} (1 - B_t) \right),$$

which is just Eq. (4) restated. We nominally use $\gamma = 10^{-4}$ and take it up to $\gamma = 10^{-3}$ for highly dynamic scenes. The terms in parentheses that involve $\alpha_s$, $\beta_s$, and $B_t$ correspond to the predictive likelihood, which intuitively measures how accurately the prior model (the Beta distribution) accounts for the incoming photon-detection distribution. The forecaster whose initialization time is closest to the previous abrupt change (or changepoint) will have the highest forecaster value—up until the next abrupt change comes along.

The newly initialized forecaster $\tilde{\nu}$ is included if $\tilde{\nu} > \operatorname{argmin}_s \nu_s$ and discarded otherwise—thus, at any given time instant, only $S$ forecasters are held in memory. Finally, using the forecasters $\{\nu_s\}$, we estimate the run length $r_t$ as described in Eq. (3). While we can use the run length to drive the adaptive smoothing parameter $\omega_t$ from here, we find it beneficial to first "min pool" the run length estimates using a $5 \times 5$ or $7 \times 7$ kernel. Our intent behind this choice is to benefit from the additional information in a patch of pixels (over individual pixels) and encourage run lengths to be conservative at a pixel if its neighborhood witnesses an abrupt change.

Algorithm 1 provides a detailed listing of our proposed adaptive-exponential integrator.

## Secs 3.2, 3.3: Zero-Order Hold Discretization

In this section, we derive the discrete state-space model as it arises from applying zero-order hold discretization to the underlying continuous-time system. We assume that the underlying continuous-time system to the selective state-space model described by Eq. (5) is

$$\frac{dh(t)}{dt} = -f(u(t))h(t) + g(u(t))u(t). \quad (11)$$

Now assuming zero-order hold discretization, which implies that the input $u(t)$ is constant between sampling instances, say $(t_i, t_j]$:

$$\frac{dh(t)}{dt} = f(u(t_j))h(t) + g(u(t_j))u(t_j), \quad (12)$$

which yields the update

$$h(t_j) = e^{-f(u(t_j))(t_j - t_i)} h(t_i) + (1 - e^{-f(u(t_j))(t_j - t_i)}) g(u(t_j)) u(t_j).$$

Renaming variables, we can discretize Eq. (11) as

$$h_t = a_t h_{t-\Delta t} + b_t u_t, \quad (13)$$

where,

$$a_t = e^{-f(u_t)\Delta t}, \quad b_t = (1 - a_t) g(u_t). \quad (14)$$

Notice that we can also choose to obtain an update rule for $h_{t+\Delta t'}$ from $h_{t-\Delta t}$, which we do when altering inference rates of the SSMs, as long as $u_t$ is constant between $(t - \Delta t, t + \Delta t']$ (implying that the zero-order hold assumption persists). Since our sampling of the input feature maps is event-driven, *i.e.*, based on substantial changes in the input, this is a valid assumption if our event-driven thresholds are not too large.

## Sec 4: Inference Mode Specifications

In this section, we provide more implementation details regarding the efficient QNN inference modes proposed in Sec. 4.

### Eventful Computation

Gating layers operate slightly differently between convolutional and transformer layers. For convolutional layers—which operate on tensors $u_t \in \mathbb{R}^{(\mathtt{H},\mathtt{W},\mathtt{C})}$, *i.e.*, a 3-dimensional time-varying tensor with height, width, and channel dimensions—gating first subtracts the current tensor from a previously stored reference value:

$$\Delta u_t \leftarrow u_t - u_{\text{ref}}. \quad (15)$$

Then, the event policy is applied on $\Delta u_t$, *e.g.*, a policy that selects the top-$K$ fraction of $u_t$ values. Then the reference is updated for the changes that are indeed propagated:

$$u_{\text{ref}} \leftarrow u_{\text{ref}} + \text{EventPolicy}(\Delta u_t). \quad (16)$$

The first time eventful compute is used, the reference tensor is set to zero and all values of $u_t$ are propagated. In other words, the first step involves dense operations. Further, note that gating for the convolutional layer exploits sparsity across both channel and spatial dimensions.

For transformers, we work with 2-dimensional tensors $u_t \in \mathbb{R}^{(L,C)}$, where $L$ is the token length dimension, which can vary depending on the resolution of inputs that are processed by the (spatial) transformer layer. For instance, when working with vision transformers that involve an initial patch-embedding stage that breaks a $256 \times 512$ image down to patches of $16 \times 16$ pixels, the token length would be 512. Notably, vision transformer have a variety of operations that are applied token-wise, including the token-wise multi-perceptron layer (MLP) and query-key-value generation. We adopt the design of Dutson et al. [14] and apply gating for token-wise operations using a gather and scatter routine. The overall operation is quite similar to Eqs. (15) and (16), except that rather than work with sparse tensors $\Delta u_t$, the non-zero values are packed into a smaller dense tensor (gather operation). After computing relevant token-wise blocks,

**Algorithm 1** Adaptive-exponential smoothing. Typical ranges for the decay factor $\gamma$ are $[10^{-6}, 10^{-4}]$, with larger values resulting a more responsive motion modeling, and thus, less blur. Further, we nominally set $S = 10$ and $k = 5$.

```
Require: Quanta sensor response, $B_t(\mathbf{p})$
         Decay factor, $\gamma$
         Number of forecasters, $S$
         Pixel locations, $\mathcal{P}$
         Total bit-planes, $T$
         Min-pool kernel size, $k$
 1: function AdaptiveExponentialSmoothing($B(\mathbf{p}, t), \gamma$)
 2:     Forecasters, $\nu_s(\mathbf{p})$ with $1 \leq s \leq S$.
 3:         $\nu_1(\mathbf{p}) \leftarrow 1, \nu_s(\mathbf{p}) \leftarrow 0$ for all $s > 1, \mathbf{p} \in \mathcal{P}$.
 4:     Forecaster time instants, $\mathcal{T}_s(\mathbf{p})$ with $1 \leq s \leq S, \mathbf{p} \in \mathcal{P}$.
 5:         $\mathcal{T}_1(\mathbf{p}) \leftarrow 1, \mathcal{T}_s(\mathbf{p}) \leftarrow 0$ for all $s > 1, \mathbf{p} \in \mathcal{P}$.
 6:     Alphas (of a Beta prior), $\alpha_s(\mathbf{p})$ with $1 \leq s \leq S, \mathbf{p} \in \mathcal{P}$.
 7:         $\alpha_1(\mathbf{p}) \leftarrow 1, \alpha_s(\mathbf{p}) \leftarrow 0$ for all $s > 1, \mathbf{p} \in \mathcal{P}$.
 8:     Betas (of a Beta prior), $\beta_s(\mathbf{p})$ with $1 \leq s \leq S, \mathbf{p} \in \mathcal{P}$.
 9:         $\beta_1(\mathbf{p}) \leftarrow 1, \beta_s(\mathbf{p}) \leftarrow 0$ for all $s > 1, \mathbf{p} \in \mathcal{P}$.
10:     Adaptive EMA, $\mathcal{I}_{\text{adapt}}(\mathbf{p}) \leftarrow 0$ for all $\mathbf{p} \in \mathcal{P}$.
11:     for $t \in \{1, \dots, T\}$ do
12:         for $\mathbf{p} \in \mathcal{P}$ do
13:             $r_t(\mathbf{p}) = \left(\sum_s \nu_s(\mathbf{p})(t - \mathcal{T}_s(\mathbf{p}))\right) \Big/ \left(\sum_s \nu_s(\mathbf{p})\right)$    ▷ run-length estimation, Eq. (3)
14:         for $\mathbf{p} \in \mathcal{P}$ do
15:             $r_{\text{min pool}}(\mathbf{p}) \leftarrow r_t(\mathbf{p})$
16:             for $\mathbf{q} \in \text{Neighborhood}_k(\mathbf{p})$ do
17:                 $r_{\text{min pool}}(\mathbf{p}) \leftarrow \min\{r_{\text{min pool}}(\mathbf{p}), r_t(\mathbf{q})\}$    ▷ Min pooling run lengths
18:         for $\mathbf{p} \in \mathcal{P}$ do
19:             $\omega_t(\mathbf{p}) = e^{-1/r_{\text{min pool}}(\mathbf{p})}$    ▷ modulating the exponential smoothing, Eq. (3)
20:             $\mathcal{I}_{\text{adapt},t}(\mathbf{p}) \leftarrow \omega_t(\mathbf{p})\mathcal{I}_{\text{adapt},t-1}(\mathbf{p}) + (1 - \omega_t(\mathbf{p}))B_t(\mathbf{p})$    ▷ recursive update, Eq. (2)
21:             $\tilde{\nu}(\mathbf{p}) \leftarrow 0$
22:             for $s$ such that $\alpha_s(\mathbf{p}) > 0$ do
23:                 Define $l_s(\mathbf{p}) = (\alpha_s(\mathbf{p})B_t(\mathbf{p}) + \beta_s(\mathbf{p})(1 - B_t(\mathbf{p})))/(\alpha_s(\mathbf{p}) + \beta_s(\mathbf{p}))$    ▷ compute predictive likelihoods
24:                 $\nu_s(\mathbf{p}) \leftarrow (1 - \gamma)\nu_s(\mathbf{p})l_s(\mathbf{p})$    ▷ update older forecasters, Eq. (4)
25:                 $\alpha_s(\mathbf{p}) \leftarrow \alpha_s(\mathbf{p}) + B_t(\mathbf{p})$
26:                 $\beta_s(\mathbf{p}) \leftarrow \beta_s(\mathbf{p}) + 1 - B_t(\mathbf{p})$    ▷ update Beta priors
27:                 $\tilde{\nu}(\mathbf{p}) \leftarrow \tilde{\nu}(\mathbf{p}) + \gamma l_s(\mathbf{p})\nu_s(\mathbf{p})$    ▷ new forecaster, Eq. (10)
28:             if $\tilde{\nu}(\mathbf{p}) > \min_s \nu_s(\mathbf{p})$ then    ▷ check if the new forecaster can replace an stale one
29:                 Denote $s_{\text{min}}(\mathbf{p}) = \text{argmin}_s\, \nu_s(\mathbf{p})$
30:                 $\nu_{s_{\text{min}}}(\mathbf{p}) \leftarrow \nu'(\mathbf{p})$
31:                 $\mathcal{T}_{s_{\text{min}}(\mathbf{p})} \leftarrow t$
32:                 $\alpha_{s_{\text{min}}(\mathbf{p})} \leftarrow 1,\ \beta_{s_{\text{min}}(\mathbf{p})} \leftarrow 1$    ▷ initialize uniform prior
33: return $\mathcal{I}_{\text{adapt}}(\mathbf{p})$
```

the updates are scattered back. To exploit eventfulness in the core computations of self-attention, we further follow the design of eventful transformers [14].

Accumulation layers, for both transformer and convolutional models, simply store the sum of changes ($\Delta u_t$) over time and transmit the summed value.

## Change-driven Inference

To keep track of significant changes, we adopt a similar mechanism as a (convolutional) gating layer. The policy, however, is not top-$K$; instead, as written in Eq. (9), we threshold the difference between $\mathcal{I}_{\text{adapt}}$ and the reference value. When the fraction of significant changes as defined by this threshold exceeds a certain value, we run the rest of the QNN stack.

## Sparse Readout

For the sensing stage, we compute output bits at each query time as the minimum transmission entailed between the dense (transmit a data point regardless of it being zero or nonzero), compressed sparse column (CSC), and compressed sparse row (CSR) formats. Additionally, we assume that each data point is represented by 10 bits and shares a 10-bit timestamp information across the encoded packet (which negligibly affects the bits/pixel/second or bps).

While not our main consideration, the sparse readout can be applied to QNN layers beyond the sensor-proximal QNN layers. For intermediate stages, which output 3-dimensional tensors (channels, height, and width dimensions), we cannot use CSC and CSR encoding—since these apply to sparse matrices only—so we resort to the more naïve coordinate-list encoding (COO). COO encodes each non-zero element and its position in the tensor. We do not use COO for sparse matrices (when encoding the readout of the sensor-proximal stage), since this format is less space-efficient than CSC or CSR.

## Summary of QNN Inference Modes

Tab. 2 provides a summary of QNN inference modes that arise from combining eventful computation and irregular sampling, as well as the sequential and parallel forms of the core state-space recurrence (Eqs. (5) and (7)) employed by QNN layers.

| First-layer sampling | Computation | Utility |
|---|---|---|
| Evenly-spaced | Parallel | Throughput (batched) |
| Evenly-spaced | Sequential | Constant memory usage |
| Evenly-spaced | Eventful | Low FLOP count |
| Change-driven | Sequential | Reduced runtime |
| Change-driven | Eventful | Reduced runtime & FLOP count |

Table 2. **QNN inference modes.** Using state-space duality that converts the recurrence of Eq. (5) to a matrix-vector product [10], we can parallelize QNNs during training. Whereas, their sequential inference features constant memory consumption. Eventful computations can significantly lower floating-point operations, but may require specialized sparse-workload accelerators. Change-driven perception, a complementary mode, runs QNNs at sporadic and irregularly-spaced instants.

**Eventful computations and change-driven sampling.** These inference modes present complementary ways of allocating compute resources according to the level of motion in a scene. Eventful computations, involving neuron-level control flow, is an enticing north star, especially as hardware support for sparse operations grows. Change-driven inference operates at the network level, providing a direct realization of run-time benefits, but with a coarser resource adaptivity. The latter's utility is conspicuous in real-world high-speed acquisition that benefits from the "temporal foveation" of change-driven sampling.

## Sec. 5.1: Quanta Neural Network Details

We now provide architecture-specific details of our proposed QNNs—the modifications involved in converting image- and video-based neural networks to quanta versions—and details specific to dataset simulation and neural-network training.

### Depth Estimation

We convert the DepthAnything-v2 model, whose objective is relative-depth estimation (*i.e.*, depth maps are estimated up to an unknown scale and shift factor) from a single image to its quanta equivalent by inserting a QNN sensing layer and QNN layers after every vision transformer (ViT [2]) block. We highlight that the DepthAnything-v2 model uses DINO-v2 [56] as a pre-trained encoder, which is a family of feature encoders that have been trained in a self-supervised manner on large corpus of data and subsequently distilled into smaller models; thus, our depth-estimation QNN case study also speaks to the broader compatibility of QNN layers with a slew of DINO-based computer vision models.

**Dataset generation.** We use Blender [9] to generate a synthetic dataset consisting of 1500 unique sequences from 30 indoor scenes. Specifically, we render about a quarter second of ground-truth frames and depth maps for each sequence at 500fps. We further interpolate the ground truth frames by a factor of $32\times$ using RIFE [31] and then sample these using the quanta-image sensor's response model described in Eq. (1) to produce binary frames at 16kHz. This yields 129 ground truth depth maps and 4097 binary frames per sequence, all at a resolution of $512 \times 512$ pixels. A single sequence takes about $3-5$ minutes to render, interpolate, and sample on a single RTX 3090, depending on the scene complexity, resulting in a total simulation time of roughly 5-GPU days.

**Training details.** We train the DepthAnything-v2 QNN on our simulated dataset for 30 epochs using the Adam optimizer [36] with initial learning rate $10^{-6}$ that is annealed to a value of $10^{-9}$ using a cosine learning-rate schedule. We also finetune the pre-trained image-based DepthAnything-v2 model on the same dataset (we train only its DPT head [58]) for our compute-performance and rate-performance comparisons.

### Point Tracking

We start with the Pips++ model [83] which tracks a set of query points throughout a video. The model first computes feature maps for each video frame using a 2D residual convnet, post which feature vectors are extracted at the query points and used to compute multi-scale correlation cost volumes. Pips++ invokes a zero-velocity assumption for the output tracks at initialization. The initialized tracks are iteratively refined along with the cost-volume correlations to yield a final trajectory estimate. For the "lifted" QNN, we focus on the feature extractor and insert a QNN sensing layer, and QNN layers after each 2D residual block. We leave the iterative track-refinement network as-is.

**Dataset generation.** We generated multi-frame point tracking data for quanta-image sensors using synthetic datasets similar to Tap-Vid [11]. We first generate high-speed sequences corresponding to the motion speed at 2000 FPS, each lasting 128 frames, with every frame labeled. We then use learned interpolators to upsample the frames and simulate binary frames. This approach provides tracking annotations for every 16 binary frames during training.

To generate tracked points, we modified the Kubric simulator [37], a physics-based engine that models object motion with forces such as gravity, friction, and restitution. We used the Movi-e subset, which features objects from Google Scanned Objects. By default, Movi-e runs at tens of frames per second. To simulate high-speed scenes, we slowed the virtual world by scaling down gravity (reducing fall speed), restitution (weakening collisions), and friction (minimizing rebound effects), setting `obj.friction=0.1`, `obj.restitution=0.2`, and `gravity=1`. This force scaling ensures that both the frames and annotations are appropriately synchronized.

In terms of motion, half of the dataset includes object motion along with randomized linear camera movement, while the other half features a static camera to isolate local motion. Kubric's default outputs do not include 2D point tracks, so we adapted processing functions from the Kubric Long-Term Point Tracking challenge to project 3D coordinates into 2D points. Each scene generates a video with $N$ randomly sampled points, tracking their (x, y) positions over time along with occlusion flags.

Finally, we apply deep interpolation using RIFE [31], increasing the frame rate by 16× while keeping annotations at their original rate. Computation occurs on all consecutive binary frames, but targets are evaluated only at annotation timestamps (every 16 binary frames). Figure 11 illustrates an example of simulated 2D tracked points and binary frames. In total, we generated 2,000 training clips (128 frames each) and 40 test/validation clips. Videos have a resolution of 256×256, and generating a 128-frame sequence takes approximately 10 minutes on an Intel Xeon Platinum 8260 CPU.

**Training details.** We train the Pips++ QNN on our point-tracking dataset for 40 epochs using the Adam optimizer with initial learning rate $10^{-4}$ that is annealed to a value of $10^{-9}$ using a cosine learning-rate schedule. We also finetune the pre-existing image-based Pips++ on the same dataset for our compute-performance and rate-performance comparisons.

### Intensity Restoration

We adopt the same video restoration architecture as Sundar et al. [67] which consists of 8 space-time factorized densely-connected blocks [71] and operates on feature maps with 64 channel dimensions. We replace the transformer modules that operate in the time domain with QNN layers of similar parameter count (identical number of heads and dimensions per head). We also replace the 3D convolutions in each dense block with

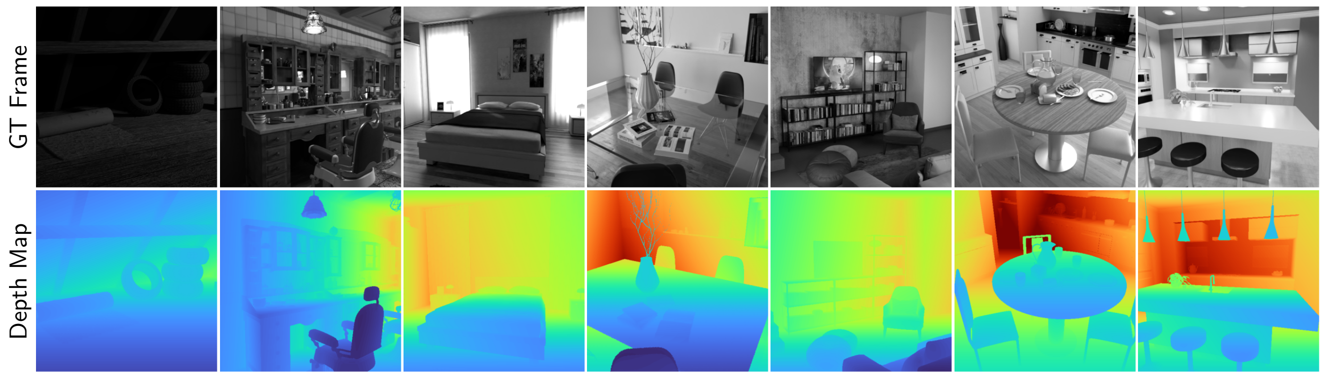


Figure 10. **Example scenes from the generated depth dataset.**

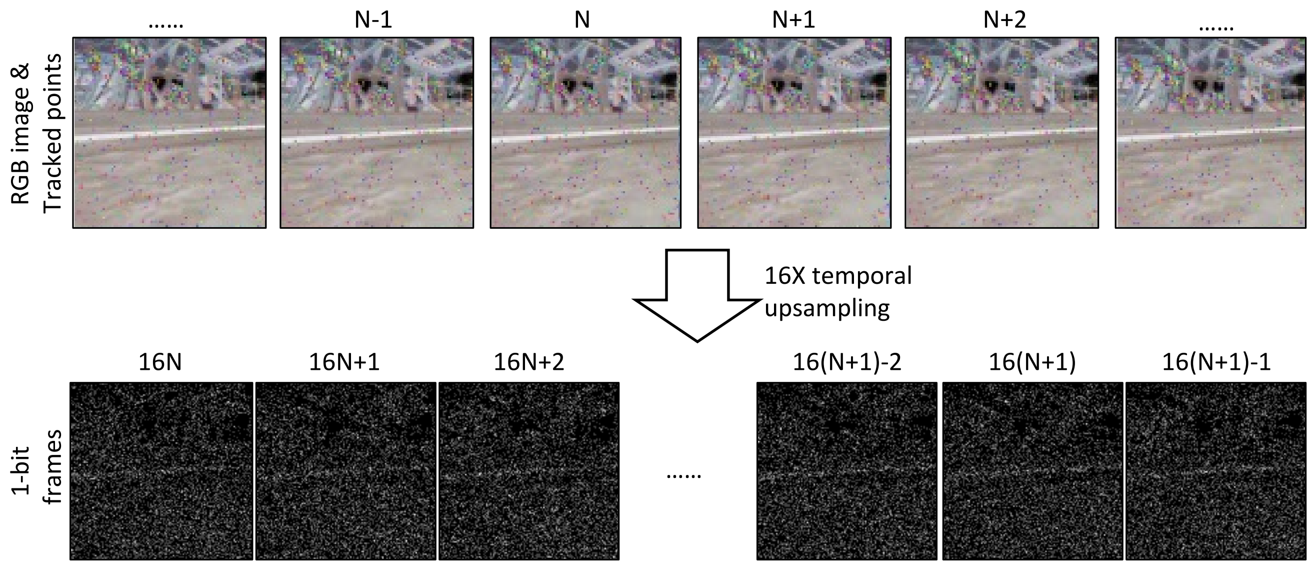


Figure 11. **Dataset samples from our generated multi-frame point-tracking dataset.**

2D convolutions, discarding convolution in the time dimension. As before, we also insert a first (sensor-proximal) QNN layer.

**Training details.** We train the intensity-restoration QNN using the XVFI dataset, from which we simulate quanta frames after interpolating the original videos from 2000 Hz to 64000 Hz using the learned RIFE model. From each interpolated video, we simulate quanta frames by treating pixel values as linear intensities and assuming an average photons-per-pixel rate that is randomly sampled in the range $(0.05, 0.5)$ (per quanta sequence). We train for 60 epochs using the Adam optimizer with an initial learning rate $10^{-4}$ that is annealed to a value of $10^{-9}$ using a cosine learning-rate schedule.

## Sec 5.1: Expanded Visualizations

Fig. 12 and Fig. 13 show multiple-time instant (multi-frame visualizations) of Fig. 5 and Fig. 7 respectively. For Fig. 6, we overlaid the cumulative tracks (*i.e.*, the trajectory of each query point) on the final frame, so we do not present a similar expanded point-tracking result here. Further, we provide more QNN inference results in Fig. 14.

In Fig. 15, we show a version of the result in Fig. 6, but with a logarithmically-scaled x-axis. We used linear scaling in the main paper to ensure consistency with Figs. 5 and 7, but the order of magnitude compute improvements over reconstruction-based quanta vision makes it difficult to discern our QNN approaches in the plot.

## Sec 5.1: Choice of Reconstruction-Based Quanta Vision Baselines

We include the following reconstruction-based methods as baselines in Sec. 5.1, by first running reconstruction and then the relevant computer-vision task:

- **Video denoising** (in Fig. 5): the network architecture is from Li et al. [41] and is trained using the interpolated XVFI dataset to denoise a stack of 16–64 frames. Each frame is the (temporal) average of 64 quanta frames. This architecture processes

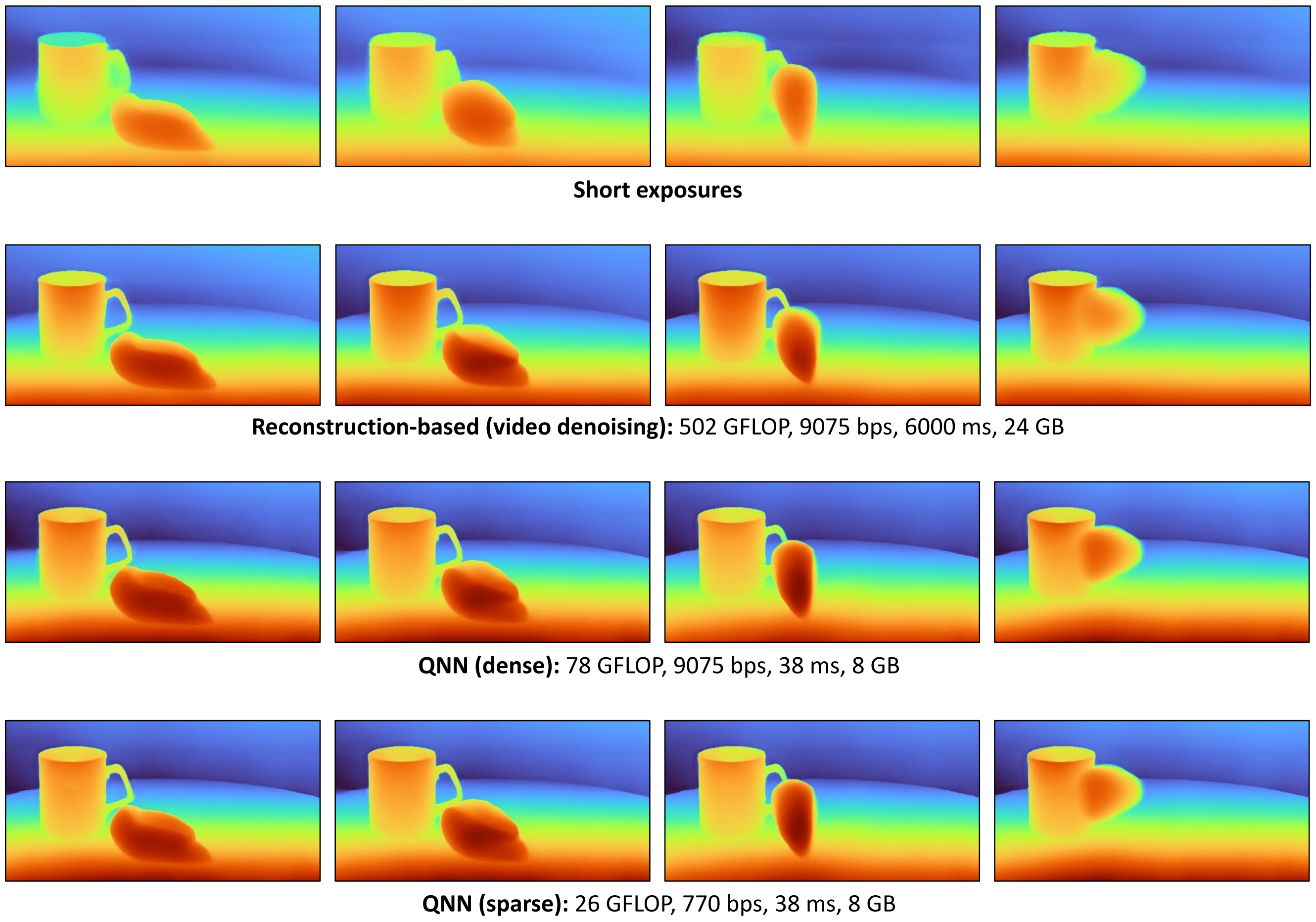


Figure 12. **Visualization of Fig. 5 across multiple time instants.** Efficiency metrics are included in the caption.

the input burst en masse, *i.e.*, not sequentially. So the latency cost for a downstream task is the cost of reconstructing *all* frames plus the latency for running the relevant task.

- **QUIVER** (in Fig. 6): is a recent burst-restoration architecture which operates recurrently. So we estimate its latency as the time taken to process a single input; in the case of QUIVER, this is the average of 80 quanta frames. (We tried to closely match the settings of Chennuri et al. [8], where the average of 8 quanta frames captured at a speed of 10 kHz is used as the input—we commensurately scale this number up for our quanta-sensor frame-rate.)
- **Bandwidth-efficient videography** [67] (in Fig. 7): a recent event-inspired videography technique for quanta image sensors that uses an architecture similar to [71] for image reconstruction from the method's eventful readout. Like the video denoising method, the reconstruction approach here is not recurrent, which leads to high latencies.

While we haven't provided a comparison to Quanta Burst Photography (QBP [51]) in the main paper, we provide a latency (time to produce the first output) comparison of our intensity-restoration QNN to QBP, using the same quanta-frame sequence as Fig. 7, in Fig. 16.

## Sec 5.1: Warmup Steps

Since QNNs are causal, they must see an initial number of photon detections before QNN layers reach a steady state and can filter out photon noise. For our results in Sec. 5.1, warmup involved 256 binary frames (or 2.64 ms). We present results and compute efficiency metrics after the warmup phase.

## Sec 5.2: CPU Run-time Speedups of Eventful Compute

We present CPU speedups when using eventful inference with a top-10% and top-20% event policy on CPU across all three QNN architectures (intensity reconstruction, depth estimation, and point tracking). For convolutional layers, we use the custom sparse and dense Cpp implementations corresponding to Dutson et al. [13]. For transformer layers, no custom backend was needed since Pytorch includes gather and scatter primitives. As

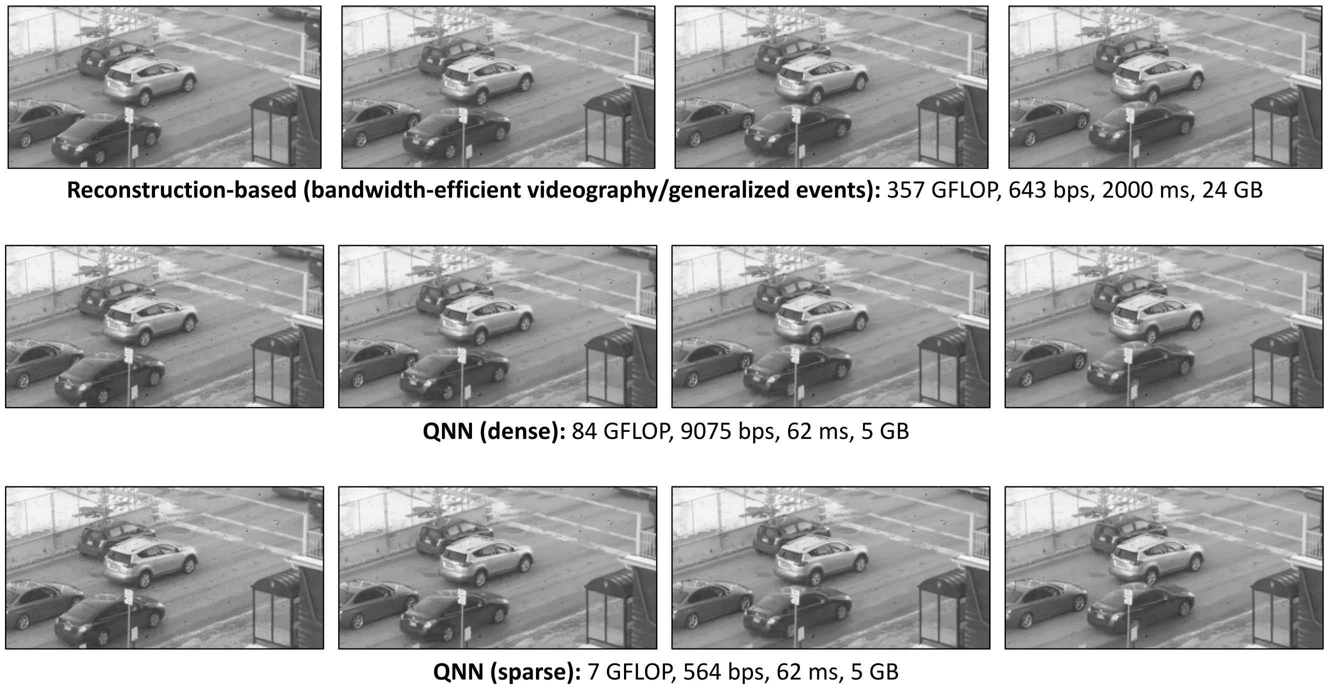

**Reconstruction-based (bandwidth-efficient videography/generalized events):** 357 GFLOP, 643 bps, 2000 ms, 24 GB
**QNN (dense):** 84 GFLOP, 9075 bps, 62 ms, 5 GB
**QNN (sparse):** 7 GFLOP, 564 bps, 62 ms, 5 GB

Figure 13. **Visualization of Fig. 7 across multiple time instants.** Efficiency metrics are included in the caption.

seen in Tab. 3, we see a 2–4× speedup over dense computations when resorting to eventful or sparse computations.

## Sec 5.2: Rate-Distortion Analysis of Intensity-Restoration QNN

Fig. 18 presents a rate-distortion analysis that compares the QNN-based video reconstruction to a quanta restoration technique, Generalized Events [67], that aims to reduce sensor readout—and so is closest to the QNN approach along the (readout) efficiency axis. The QNN uses a similar restoration architecture as the video-restoration network used in Generalized Events, but replaces the time-domain non-causal Transformer-based decoder with *causal QNN layers*, thereby achieving an order of magnitude compute reduction, with minimal loss in reconstruction quality (Fig. 18). There are other intensity reconstruction approaches (*e.g*., Chennuri et al. [8], Ma et al. [51]) which require reading off the entire sensor data, and are thus prohibitively expensive both from bandwidth and compute perspectives.

## Sec 5.3: Change-driven Sampling

We provide an additional example of change-driven sampling in Fig. 17, where we run our DepthAnything-v2 and intensity-restoration QNN irregularly. Unfortunately, we cannot run our Pips++ QNN irregularly without making modifications to its iterative track-refinement module, which is based on temporal 1D convolutions, to work on irregularly sampled feature maps. One way to do this would be to incorporate continuous-time models, such as the state-space models QNN layers use, for its track refinement network, but we leave this for future work.

## Comparison to Direct-Perception Methods

Goyal and Gupta [22] consider SPAD-based quanta sensors that run at high speeds, but assume no motion; nonetheless, we compare against this method on depth estimation. We finetune the DepthAnything-v2 model using the photon-space consistency loss (stack of sums of 16, 64, 256 quanta frames). We run inference on the sum of 256 quanta frames, which provided the best noise-blur tradeoff for the scene shown in Fig. 19.

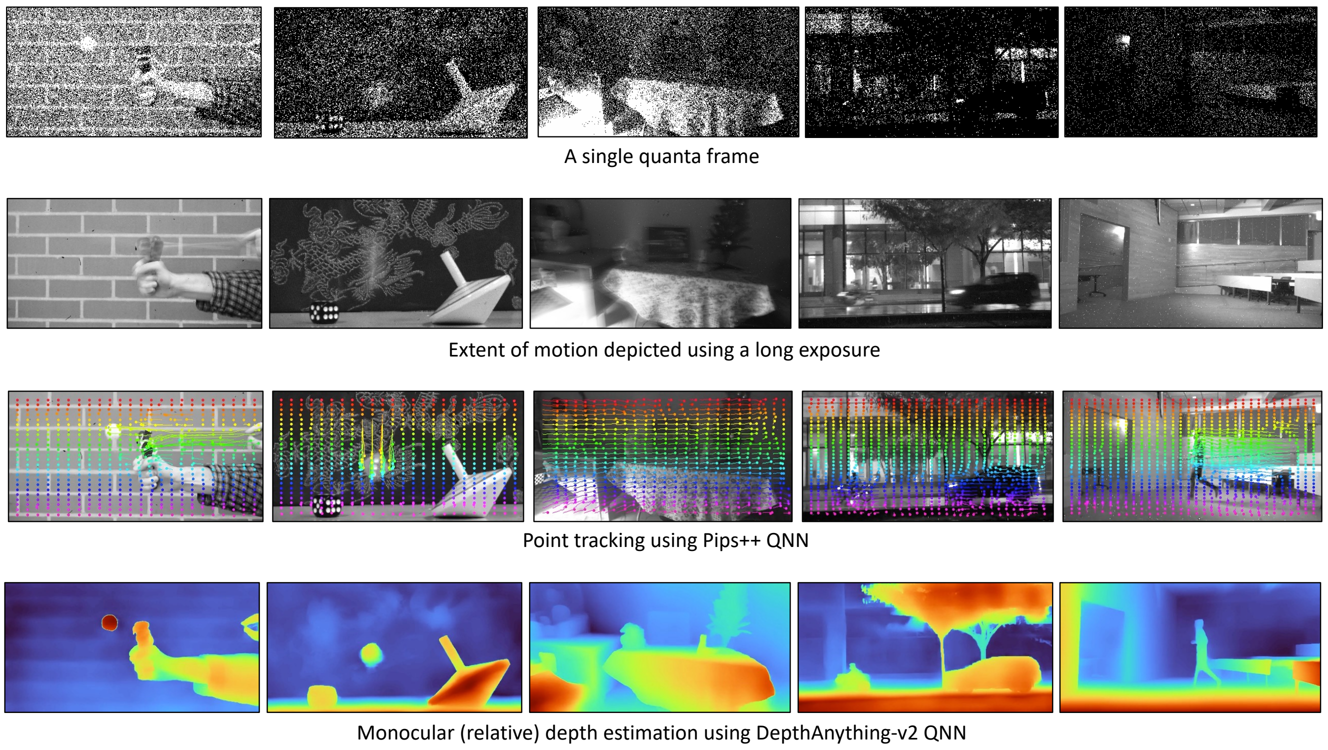

A single quanta frame
Extent of motion depicted using a long exposure
Point tracking using Pips++ QNN
Monocular (relative) depth estimation using DepthAnything-v2 QNN

Figure 14. **More results.** Point-tracking results are overlaid here on the reconstruction results obtained using our intensity-restoration QNN.

| Task | Latency, Speed up | | |
|---|---|---|---|
| | Dense | Eventful top-20% | Eventful top-10% |
| Intensity recons. | 44s, $1\times$ | $20s$, $2.2\times$ | $13.2s$, $3.3\times$ |
| Depth estimation | 12s, $1\times$ | $4.8s$, $2.5\times$ | 3s, $4\times$ |
| Point tracking | 86.5s, $1\times$ | 43.6s, $2\times$ | 24s, $3.6\times$ |

Table 3. **CPU speedups**. We observe a 2–4$\times$ speedup using our CPU implementation of sparse operations (against a theoretical FLOP reduction of 5–10$\times$. The latency numbers for QNNs are higher for the point-tracking task compared to other tasks since Pips++ estimates trajectories in a sliding-window manner: using non-overlapping windows of 16 feature vectors.

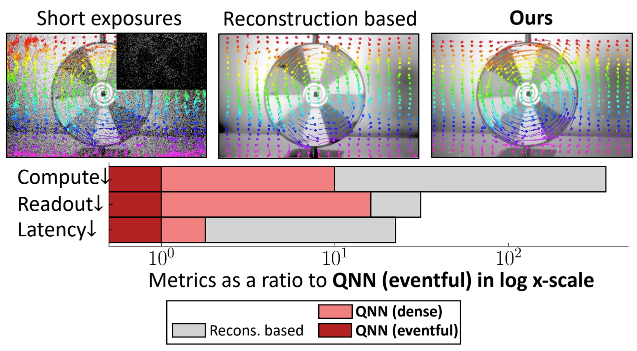


Figure 15. **Results from Fig. 6 plotted with a log-scaled xaxis.**

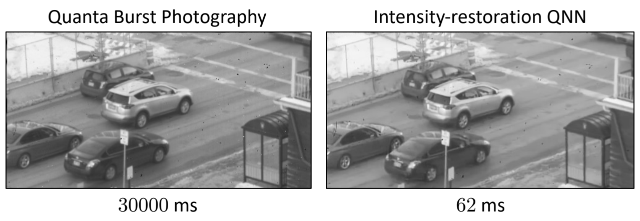
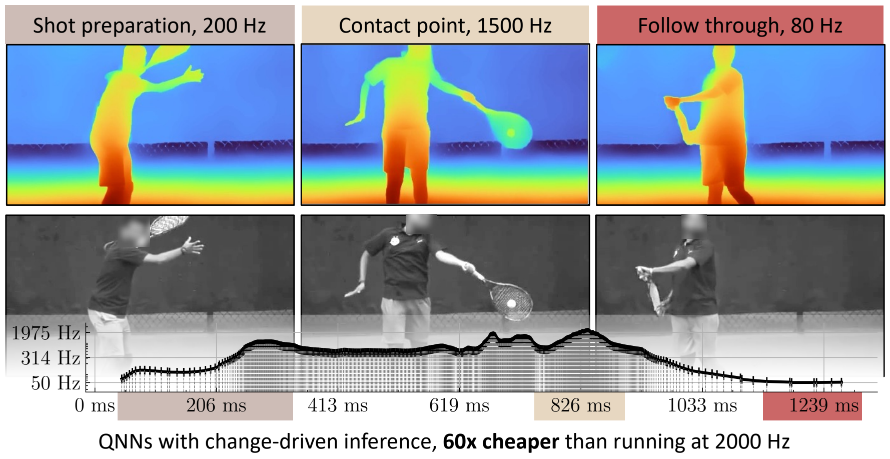


Figure 16. **Latency comparison to Quanta Burst Photography [51].** Our intensity-restoration QNN is 483$\times$ faster in producing a single output.



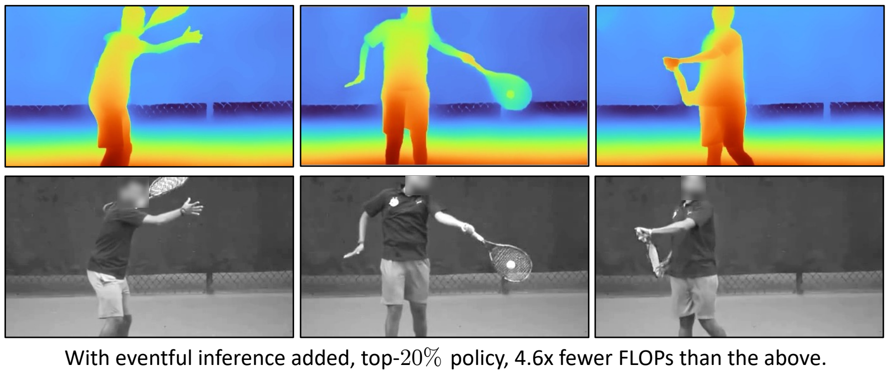

With eventful inference added, top-20% policy, 4.6x fewer FLOPs than the above.

Figure 17. **Irregular, change-driven sampling applied to a tennis forehand sequence for depth estimation and intensity restoration**. We see three distinct speed phases emerge from the shot making. Change-driven sampling is about 60× cheaper in run time, FLOPs and readout that running at the highest possible speed in this sequence (which is around ~2000 Hz). With eventful computations, we can further reduce FLOP counts—here by 4.6× when using a top-20% event policy.

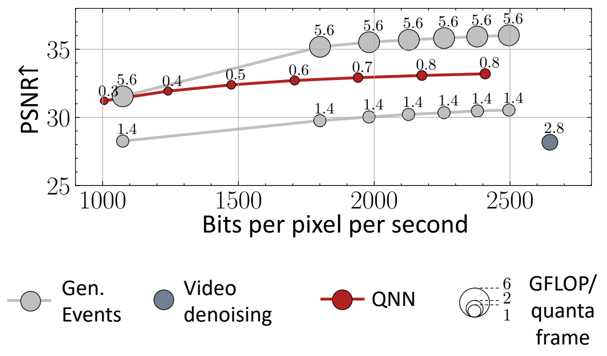


Figure 18. **Rate-distortion tradeoff on intensity restoration using the i2k dataset [8].** There is a 1–2 dB PSNR gap to a reconstruction-based approach (Gen. Events). However, QNNs are 7–18× cheaper FLOP-wise. Fewer frames may be reconstructed by Generalized Events to lower FLOP counts, but this leads to worse performance than ours. We report GFLOP per quanta-frame since we reduce the number of frames decoded by Generalized Events (by reducing the number of frames that are reconstructed).

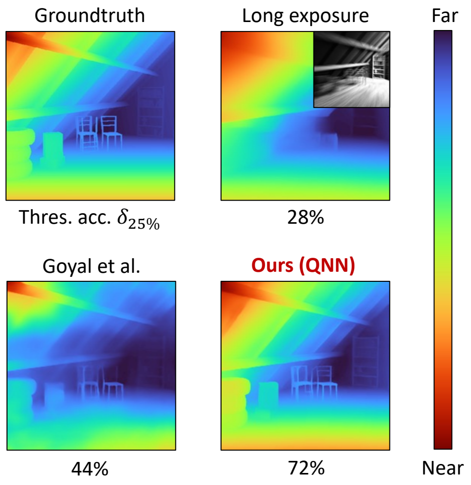


Figure 19. **Comparison to Goyal and Gupta [22].** The method does not handle motion and is subject to the noise-blur tradeoff.