SpiderCam: Low-Power Snapshot Depth from Differential Defocus

Supplementary Material

S1. Implementation Details

In this section, we provide implementation details of the five steps to the SpiderCam algorithm.

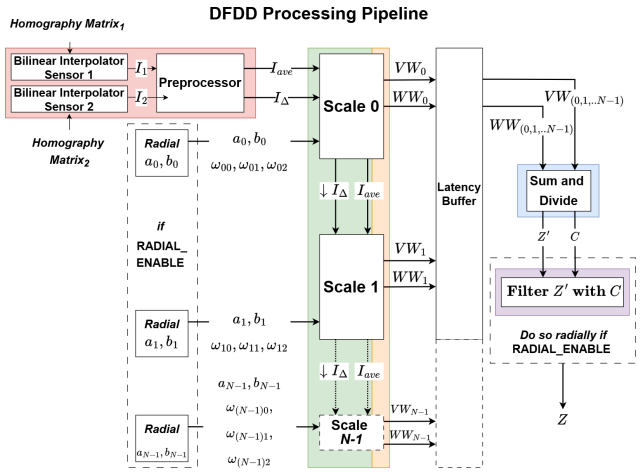

Figure S1. General overview of the algorithm.

S1.1. Homography

We apply an affine transformation homography using bilinear interpolation to the images to ensure sub-pixel alignment. A purely mechanical solution to align the images would be expensive and difficult to do due to the high degree of precision required. The closer focal plane image is known as I_1, and the further focal plane image as I_2.

S1.2. Preprocessing

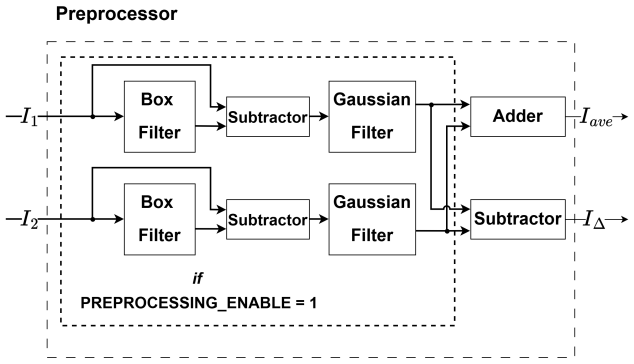

Figure S2. Preprocessing module.

In the preprocessing step, to denoise the image, we apply a box filter followed by subtraction and then a Gaussian filter. This step only applies if `PREPROCESSING_ENABLE` is set to one. Otherwise, we skip directly to calculating I_{ave} and I_Δ, which are simply the sum and difference of I_1 and I_2:

$$I_{ave} = I_1 + I_2, \qquad I_\Delta = I_1 - I_2$$

S1.3. N^{th} Scale

For a particular scale N, we define:

$$V_N = a_n \times \texttt{laplacian}(I_{ave})$$

$$W_N = b_n \times V_N - I_\Delta$$

Then, without additional spatial derivatives:

$$VW_N = w_{n0} \times V_N \times W_N$$

$$WW_N = w_{n0} \times W_N \times W_N$$

With additional spatial derivatives enabled (as indicated by `DX_DY_ENABLE`), we sum the derivatives to produce VW_N and WW_n:

$$\begin{aligned} VW_N = {} & w_{n0} \times V_N \times W_N \\ & + w_{n1} \times V_N\, dx \cdot W_N\, dx \\ & + w_{n2} \times V_N\, dy \times W_N\, dy \end{aligned}$$

$$\begin{aligned} WW_N = {} & w_{n0} \times W_N \times W_N \\ & + w_{n1} \times W_N\, dx \cdot W_N\, dx \\ & + w_{n2} \times W_N\, dy \times W_N\, dy \end{aligned}$$

S1.4. Summation and Division

The final step is to sum the respective VWs and WWs from each scale and divide, to give the depth and confidence values:

$$Z' = \frac{\sum_{i=0}^{N-1} VW_N}{\sum_{i=0}^{N-1} WW_N}, \qquad C = \sum_{i=0}^{N-1} WW_N$$

The resulting Z' in then filtered using the confidence value and working range values to produce the final depth map Z.

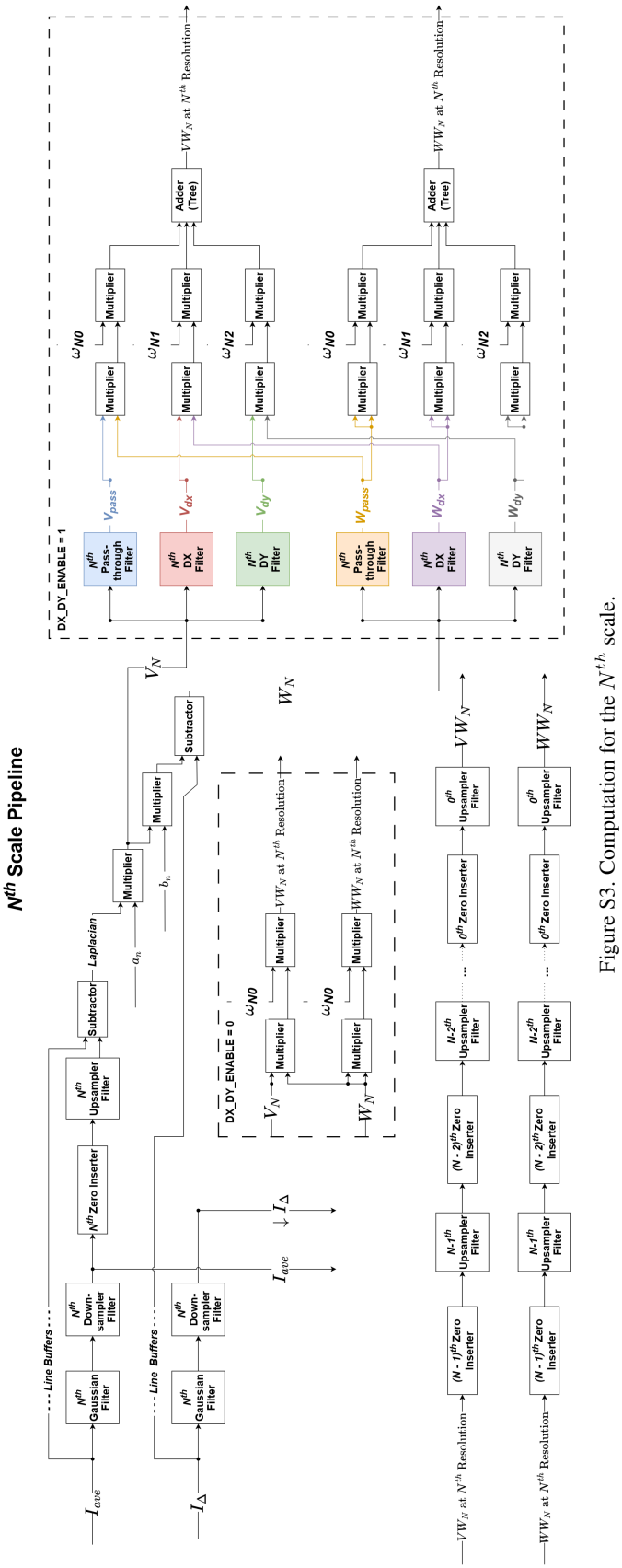

Figure S3. Computation for the N^{th} scale.

S1.5. Radial Zone Adaptation

When `RADIAL_ENABLE` is active, the logic for per-zone parameter selection operates as follows:

Algorithm 2 Spatial variation. Zone-based parameter evaluation uses squared distance to avoid hardware square-root operations and tracks pixel streaming to save memory.

Input: pixel's distance from center, $streaming\_distance\_square$,
per zone and/or per scale values,
$r^2_{zones}[\,], a'_{scales}[\,], b'_{scales}[\,], C'_{thresh}[\,], Z'_{min}[\,], Z'_{max}[\,]$
Output: parameters $(\{a_0, ..., a_{N-1}\}, \{b_0, ..., b_{N-1}\}, \{\omega_0, ..., \omega_{(3N-1)}\}, C_{thresh}, Z_{min}, Z_{max})$
for z in $\{zones - 1, zones - 2, ..., 0\}$ **do**
  if $streaming\_distance\_squared < r^2_{zone}[z]$ **then**
    for $scale$ in {0,1,..,$N-1$} **do**
      $a_{scale} \leftarrow a'_{scales}[z]$
      $b_{scale} \leftarrow b'_{scales}[z]$
    end for
    $C_{thresh} \leftarrow C'_{thresh}[z]$
    $Z_{min} \leftarrow Z'_{min}[z]$
    $Z_{max} \leftarrow Z'_{max}[z]$
  end if
end for

A single confidence minimum value can be used to filter the depth map but due to the Petzval field curvature, exacerbated by using a small focal length and a very small image sensor, our working range becomes quite limited. To handle this, we assign a set of parameters not only for each scale, but also for each **radial zone** that the pixel belongs to. Using the pixel's (x, y) coordinate, the $streaming\_distance\_squared$ can be computed, which is simply the distance squared of the pixel from the optical center of the image.

S2. Image Stream Processing in Hardware

In hardware, to process images efficiently, we adopt a *data streaming model*. This is a standard approach for maximizing *data locality* and works well with our algorithm since the computations apply only to local pixels. Instead of buffering an entire image frame, each convolution filter requires the kernel's height minus one lines to be buffered.

S2.1. Use of Efficient Gaussian Kernel

Our Gaussian kernel is the 5×5 Anderson–Burt Gaussian kernel [9]:

$$\frac{1}{256}\begin{bmatrix}1 & 4 & 6 & 4 & 1\\ 4 & 16 & 24 & 16 & 4\\ 6 & 24 & 36 & 24 & 6\\ 4 & 16 & 24 & 16 & 4\\ 1 & 4 & 6 & 4 & 1\end{bmatrix}$$

which can also be expressed in its linearly separable form as:

$$\frac{1}{16}\begin{bmatrix}1 & 4 & 6 & 4 & 1\end{bmatrix} \otimes \frac{1}{16}\begin{bmatrix}1\\ 4\\ 6\\ 4\\ 1\end{bmatrix}$$

This kernel is both compact—using small integer coefficients that are powers of two—and linearly separable, making it highly efficient for implementation in hardware.

S3. Hybrid Fixed Point to Efficient Floating Point Model

To minimize resource usage, the homography and preprocessing stages are implemented using *fixed-point arithmetic*. In particular, both the box filter and Gaussian filter are executed without trimming bits, thereby avoiding quantization issues. After these stages, the results are converted to floating point before computing I_{ave} and $I_{\triangle}$.

On the floating-point side, two main strategies make the design feasible:

- Use of **half precision (FP16)** instead of full precision (FP32)
- **Dropping subnormal number support**

Performing floating-point arithmetic with subnormal number support makes multipliers and dividers nearly as expensive as fixed-point arithmetic, with only a small additional cost for exponent handling. With subnormal support disabled, we can assume that the most significant bit (MSB) of both operands is always one. This guarantees that the MSB of the result will appear in either the highest or second-highest bit position, eliminating the need for an expensive variable shifter during multiplication or division.

Our adders still require the variable shifters and priority encoders associated with floating-point arithmetic, but some of the surrounding logic is simplified by removing subnormal handling. The primary reduction in resource usage for our floating-point adders comes from using half-precision (FP16) representation.

S4. The General Upsampling Problem

Our algorithm requires the generation of a *Laplacian Pyramid* in the N^{th} Scale Pipeline. Downsampling an image is straightforward; however, a naïve implementation of upsampling using bilinear interpolation would require buffering the downsampled image, which corresponds to one-quarter the number of lines of the original image. This occurs because pixels that are reverse-mapped from the upsampled image are far apart but refer to the same neighboring pixels in the downsampled image. This can be described as:

$$f(x, y) = g(x//2,\, x//2 + 1,\, y//2,\, y//2 + 1)$$

A more efficient solution leverages the fact that the downsampled image dimensions are halved. We can therefore push out a pixel from the downsampled image every four clock cycles for each clock cycle in the upsampled image. This reduces the number of lines that must be buffered to one-half of the downsampled image, which corresponds to one-eighth the number of lines of the original image. Some buffering is still required, since every other line in the upsampled image uses the same neighboring pixels from the downsampled image. Although buffering one-eighth is an improvement, it remains unfeasible (for example, one-eighth of a 480-line image is 60 lines) and scales undesirably with the image height.

S4.1. Proposed Solution

First, a clarification on how the convolutions are performed. For odd-sized kernels, which are the majority of our kernels, we crop evenly from all sides of the image to retain the original image dimension.

For even-sized kernels, due to ambiguity of cropping, we specify which border we crop from to get back to the original image dimension. In hardware, we don't actually do cropping but instead adjust the 'center' of the kernel to achieve the same behavior. We define 'br' as the bottom and right crop, and 'tl' as the top and left crop.

We solve the problem generally by avoiding the need to buffer half an image. Instead, we double the kernel sizes and interleave their coefficients with zeros in conjunction with inserting zeros into the image itself. This can be summarized using the following formulations.

Naïve approach:

1. Image is convolved with a 2×2 box filter, known as our 'Downsampler Filter'.
2. Skip every odd row and column.
3. The resulting downsampled image is then upsampled using buffering and a bilinear interpolation.

Buffering requirement: one-quarter of the image lines.

Our approach:

1. Image is convolved with a 2×2 box filter, known as our 'Downsampler Filter'

2. Insert zero at the pixels with odd row and column coordinates (zero insertion).
3. The zero-inserted image is convolved with a bilinear interpolation kernel, known as our 'Upsampler Filter'.

With the bilinear interpolation kernel specified, the upsampling process can be expressed as:

$$\text{Image} * \begin{bmatrix} \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} \end{bmatrix}_{\text{tl}} \rightarrow \text{zero insert}$$
$$\rightarrow * \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 1 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix} \rightarrow \text{upsampled image}$$

Because the box filter is even-sized, it introduces a *half-pixel shift* with respect to the coordinate space of I_Δ. We correct this shift by convolving further with a half-pixel-shifted kernel using the *opposite cropping side*:

$$\text{Image} * \begin{bmatrix} \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} \end{bmatrix}_{\text{tl}} \rightarrow \text{zero insert}$$
$$\rightarrow * \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 1 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix}$$
$$\rightarrow * \begin{bmatrix} \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} \end{bmatrix}_{\text{br}} \rightarrow \text{upsampled image}$$

The convolution of the two kernels,

$$\begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 1 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix} * \begin{bmatrix} \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} \end{bmatrix},$$

produces:

$$\begin{bmatrix} 1 & 3 & 3 & 1 \\ 3 & 9 & 9 & 3 \\ 3 & 9 & 9 & 3 \\ 1 & 3 & 3 & 1 \end{bmatrix}_{\text{br}} \times \frac{1}{16}$$

which is *linearly separable* into:

$$\begin{bmatrix} 1 & 3 & 3 & 1 \end{bmatrix}_{\text{br}} \times \frac{1}{4} \times \begin{bmatrix} 1 \\ 3 \\ 3 \\ 1 \end{bmatrix}_{\text{br}} \times \frac{1}{4}.$$

Buffering requirement: three lines of the image.

Now with this solution, at the 0^{th} scale, it is simple: we downsample, then upsample, and proceed as normal. But for the 1^{st} scale, where $\downarrow_1 I_{ave}$ and $\downarrow_1 I_\Delta$ are used, we must interleave the kernels with zeros to match the appropriate dimensions.

For example, the N^{th} Anderson–Burt Gaussian kernel across scales is given below.

0^{th} scale

$$\frac{1}{16}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

1^{st} scale

$$\frac{1}{16}\begin{bmatrix} 1 & 0 & 4 & 0 & 6 & 0 & 4 & 0 & 1 \end{bmatrix}$$

$2nd$ scale

$$\frac{1}{16}\begin{bmatrix} 1 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 6 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The N^{th} zero-inserter follows:

$$f(x,y) = \begin{cases} \text{keep original pixel, if } x \bmod 2^{(N^{th}+1)} = 0\,, \\ \text{and } y \bmod 2^{(N^{th}+1)} = 0 \\ 0, \text{ otherwise.} \end{cases}$$

The power-of-two modulus makes this operation straightforward to implement in hardware.

To then reconstruct the full-resolution image, chain zero-inserters and upsamplers from each scale back to the original size, as shown in Fig. S3.

S5. FLOPs per Pixel

All the Nth filters are equivalent besides the interleaved zeros, which make them sparse kernels to match their scale dimension. This means that the adder tree sizes and number of multipliers between scales are the same, since all the zero values can simply be optimized away.

As discussed previously, the majority of the kernels are linearly separable and consist of coefficients that are powers of two. Therefore, in the following calculations:

- A multiply by a power of two is denoted as an **easy multiply**.
- A multiply requiring a full multiplier is denoted as a **true multiply**.

Here, although N refers to the total number of scales, it can also refer to a particular scale if noted within the context of a scale and "N^{th}" always refers to a particular scale (starting from 0). Please refer to Fig. S3 for visual reference and tables Tab. S1 to see FLOPs per Filter/Operation and Tab. S2 Aggregate costs for N Scales.

Computation Breakdown per Scale

An N^{th} scale requires:

- $2\times$ Gaussian Filter
- $2\times$ Downsampler Filter
- $1 + 2\times$ Upsampler Filter
- V_N & W_N

Filter / Operation	Adders	True Mult.	Easy Mult.	Dividers	Total FLOPs
Gaussian 5×5 Filter	8	2	8	0	18
Downsampler Filter	2	0	4	0	6
Upsampler Filter	6	4	4	0	14
V_N & W_N	2	2	0	0	4
Cross Mult. w/ Weights (DX_DY_ENABLE = 1)	8	12	10	0	30
Cross Mult. w/ Weights (DX_DY_ENABLE = 0)	0	4	0	0	4
Add and Divide	$2 \times (N-1)$	0	0	1	$2 \times (N-1)+1$

Table S1. FLOP counts and arithmetic breakdown per operation. N refers to the number of scales. Total FLOPs is the sum of all arithmetic operations per filter or stage.

- $1\times$ Cross Multiplication with Weights (depending on DX_DY_ENABLE)

S6. Memory Usage

The number of image lines that must be buffered depends on the kernel height and the amount of zero interleaving at each scale. Here are the line-buffer equations for each buffer type at an N^{th} scale (where N^{th} = 0, 1, 2, ... N-1). G_h, D_h, U_h, and P_h denote the base kernel heights of the Gaussian, Downsampler, Upsampler, and Pass/DX/DY filters, respectively.

N^{th} Scale Buffer Equations

$$N^{th} \text{ Gaussian } (5 \times 5): \quad (G_h - 1)\, 2^{N^{th}} = 2^{N^{th}+2}$$

$$N^{th} \text{ Downsampler } (2 \times 2): \quad 2^{N^{th}}$$

$$N^{th} \text{ Upsampler } (4 \times 4): \quad 3 \times 2^{N^{th}}$$

$$N^{th} \text{ Pass/DX/DY } (3 \times 3): \quad 2^{N^{th}+1}$$

N^{th} Scale Buffers, $B(N^{th})$

An N^{th} scale requires these many buffers:

- $4\times$ N^{th} Gaussian buffers
- $4\times$ N^{th} Downsampler buffers
- $3\times$ N^{th} Upsampler buffers
- $2\times$ N^{th} Pass/DX/DY buffers (only if DX_DY_ENABLE = 1)
- $2 \times \sum_{i=0}^{N-1} \big[\min(i,1) \times (i-1)^{th}$ Upsampler buffer$\big]$

Which simplifies to:

- $33 \times 2^{N^{th}} +$ $2 \times \sum_{i=0}^{N-1} [\min(i,1) \times (i-1)^{th}$ Upsampler buffer$]$ (DX_DY_ENABLE = 1)

- $29 \times 2^{N^{th}} +$ $2 \times \sum_{i=0}^{N-1} [\min(i,1) \times (i-1)^{th}$ Upsampler buffer$]$ (DX_DY_ENABLE = 0)

Since each scale has a different number of buffers, and higher scales begin processing only after receiving their downsampled I_{ave} and I_{Δ}, they reach the latency buffer at different times. Thus, the latency buffer must store data equal to the longest latency path (largest scale) minus the shortest latency path (lowest scale), multiplied by $(N-1)$ scales.

N^{th} Scale Latency, $L(N^{th})$

An N^{th} scale latency amounts to these number of buffers:

- $1\times$ Gaussian buffer
- $1\times$ Downsampler buffer
- $1\times$ Upsampler buffer
- $1\times$ Pass/DX/DY buffer (if DX_DY_ENABLE = 1)
- $\sum_{i=0}^{N-1} \big[\min(i,1) \times (i-1)^{th}$ Upsampler & Downsampler Buffer$\big]$

Which simplifies to,

- $10 \times 2^{N^{th}} +$ $\sum_{i=0}^{N-1} [\min(i,1) \times (i-1)^{th}$ Upsampler & Downsampler buffer$]$ (DX_DY_ENABLE = 1)

- $8 \times 2^{N^{th}} +$ $\sum_{i=0}^{N-1} [\min(i,1) \times (i-1)^{th}$ Upsampler & Downsampler buffer$]$ (DX_DY_ENABLE = 0)

The latency buffer therefore needs to buffer:

$$(N-1) \times \big(L(N-1) - L(0)\big)$$

lines.

See Tab. S3 for the aggregate number of buffered lines across scales.

S7. Sensor Selection

To maintain low power usage, we selected the Himax HM0360 Ultra Low Power sensor. It is quite small, measuring at 2.88 mm across its diagonal and with a pixel pitch

N (Scales)	DX_DY_ENABLE	Adders	True Mult.	Easy Mult.	Dividers	Total FLOPs
1	0	28	14	28	1	**71**
1	1	36	22	38	1	**97**
2	0	70	36	64	1	**171**
2	1	86	52	84	1	**223**
3	0	124	66	108	1	**299**
3	1	148	90	138	1	**377**

Table S2. Aggregate per-pixel operation counts across N scales using the per-operation table and the updated per-scale rule (2× Gaussian, 2× Downsampler, $(1 + 2 \times N^{th})$ Upsamplers per scale, and 1× Cross Mult. with Weights per scale),V_N & W_N plus one final Add and Divide.

N (Scales)	DX_DY_ENABLE	Scale Buffers	Latency Buffers	Total Buffers
1	0	29	0	29
1	1	33	0	33
2	0	93	12	105
2	1	105	14	119
3	0	227	36	263
3	1	255	42	297

Table S3. Aggregate buffered line requirements for $N = 1$–3 scales, including buffers used in scales, buffers consumed by Latency Buffer, and Total buffers used. Each buffer is the size of one line of the image in FP16. For both DX_DY_ENABLE modes.

of 3.6 μm. We calculated that running at 32.5 FPS, the sensor only draws between 10-20 mW. The tradeoffs are the increased amount of noise in the image and using a smaller focal length to maintain a similar FoV. Using a shorter focal length creates a stronger field curvature, which we fix with our spatially varying parameters discussed earlier.

S8. FPGA power consumption and resource utilization estimation

Estimating power consumption for a previous work without access to its source code is difficult: the only way to achieve a precise estimate of the power consumed by an FPGA design is to simulate the design on real data and record net switching information, using these figures for calculation [5]. Although this is not feasible for many existing works as designs and data are not readily available, we can use vendor tools [34, 68] to attain a reasonably accurate power estimate if we are provided with circuit operating frequency, resource utilization, fanout, and activity factor [10]. While frequency and resource utilization are readily available from the papers themselves, fanout (the average number of loads driven by a driver, e.g. the average number of LUTs driven by a flip-flop) cannot be known without the actual design and activity factor (number of wire toggles between '0' and '1' per clock cycle) requires the design to be simulated on actual data.

Therefore, to get a power estimate, we choose low and high reasonable values for fanout and activity factor. For fanout, we simply choose values of 3 and 6 for low- and high- power estimates, respectively [68].

"Activity factor" (AF) refers to the number of times a logic signal toggles from '0' to '1' and back to '0' during a single clock cycle. A system's power consumption is very strongly affected by its AF; power dissipation in digital circuits largely comes from charging and discharging internal nodes. AF strongly depends on input data: if input data doesn't change from cycle to cycle, then there is no change in internal logic signals and the AF is 0%. Neglecting glitching, the worst-case AF is 50%. For random signals, the AF will be 25%. [68] recommends using an AF of 12.5% for conservatively estimating power on real-world data. In our table, we use an AF of 12.5% for our low-end power estimate and an AF of 50% for our high-end power estimate. Furthermore, we normalize all circuits to operate at 30 Mpix/s, roughly the number of pixels per second needed to process an HD video frame at 30 FPS, scaling the AF by the fraction of clock cycles that the system has to be active for to achieve 30 Mpix/s. For instance, if a work is capable of processing 100 Mpix/s, we scale the activity factor by 30 / 100, modeling reduced power consumption during inter-frame idle time due to an AF of 0%.

A more detailed version of Tab. 1 can be seen in Tab. S4.

The "Estimated Core Power" column shows power estimates reported in the original papers, where available. In the original works, these estimates were produced not by measuring the power consumption of a running system, but by feeding their design into a vendor-provided tool as described above. Because of differences in frame-rate and estimation methodology, the reported power numbers cannot be equitably compared, so we re-did the estimates as described above. Our estimates are very conservative and under-estimate the power for all except for [55].

S9. DIY Guide

Except for a few custom PCBs and 3D printed optomechanical parts, SpiderCam is entirely made up of commercial off-the-shelf components. The custom components can be cheaply and readily manufactured using PCB designs and 3D CAD files provided by us. Consult Tab. S5 for a list of all parts and how they can be acquired.

S9.1. Electrical components

Our system uses a Lattice ECP5 FPGA for its processing. We deploy our FPGA Gateware on a Lattice ECP5 Evaluation Board, part number LFE5UM5G-85F-EVN. This board carries a Lattice LFE5UM5G-85F FPGA as well as a large amount of support circuitry and peripherals. Because this evaluation board is not tailored to our specific application, we slightly modified it to reduce power consumption. We removed an unused oscillator (refdes X2), a number of unused LEDs (refdes D22-26, D31), and removed inefficient linear regulators (refdes U5-6) supplying power to unused transceivers. We used a manually soldered wire to power the transceivers from the 1.2V switching regulator (refdes U8). While these modifications save some power, they don't alter the fundamental low-power contributions: more time and a larger budget would permit the construction of a project-specific PCB (as in [45]) instead of using a modified off-the-shelf board. An entirely custom PCB would admit even greater power savings. Furthermore, to make our system work, we had to remove R99 and add a 0-Ω resistor to footprint R104 to change VCCIO6 from 3.3 V to 2.5 V.

Although the low-power image sensor that we chose for this project can be readily purchased, at the time of writing, no PCBs carrying this image sensor without integrated optics are available. To this end, we constructed a carrier board for the HM0360 image sensor which we integrated into our optomechanical assembly. The HM0360 carrier board connects to the FPGA evaluation board through a 30-pin flat-flex cable and a custom-fabricated adapter board which connects two image sensors to the FPGA evaluation board through some of its header pins. Finally, data readout is achieved via another adapter board which connects an FT232H USB-to-Parallel transceiver to header pins on the FPGA board.

Construction of all of these PCBs costs only a few hundred dollars, and copies can be ordered fully assembled from many PCB fabrication providers. KiCAD design files, which can be used to order copies, can be found at https://github.com/NUBIVlab/SpiderCam/tree/main/pcb.

Assembling the system Two image sensor boards affixed to the optical setup, as described in Sec. S9.2, need to be connected to the image sensor adapter board via flat-flex cables. The image sensor adapter board should be plugged into the FPGA development board via headers J39 and J40. An FT232H breakout board should be connected to the FPGA development board via headers J5 and J8 with the FT232H breakout adapter. A photograph of a fully assembled system can be seen in Fig. 1.

The FPGA board must be programmed with the Gateware that we developed for this project. `openocd` can be used to program the board with the appropriate Gateware images; prebuilt Gateware images can be found at https://github.com/NUBIVlab/SpiderCam/tree/main/fpga/synth/dual_scale along with programming instructions. Once the FPGA has been programmed, Python scripts at https://github.com/NUBIVlab/SpiderCam/tree/main/fpga/tools/serialcam_ft232h can be used to read data from the board using the FT232H USB-to-Parallel adapter. Explanation of the directory structure of different variants of the bit files and usage of the Python GUI is detailed at https://github.com/NUBIVlab/SpiderCam/blob/main/fpga/README.md.

S9.2. Optomechanical assembly

As described in Sec. 3.1, our system relies on taking 2 pictures of the same scene with different levels of focus. To achieve this, we use a Thorlabs CCM5-BS016 beamsplitter with a 10mm AC050-008-A-ML lens attached to each output port. Each of the HM0360 image sensor boards is aligned with the optics using a 3D printed mount; the HM0360 image sensor boards are affixed to the mounts with M1.6 bolts, and the mounts are bolted onto the beamsplitter, aligning the two image sensors with their respective lenses. One side of a partially assembled image sensor mount can be seen in Fig. S7.

Optomechanical assembly calibration In order to achieve the defocus effect, the focal distance of the two sensors must vary. We used precision shims to offset the image sensors in the direction of the lenses optical axes and screwed the lenses in by different amounts to achieve a more precise offset.

A homography calibration was also needed. The pixel pitch on our image sensors is 3.6 micrometers, but the 3D printed optical mounts are manufactured with a tolerance of about 200 micrometers. Because of this, the

Name	Type	FPGA	Resolution	Stereo Disparity Levels	Mpix/sec	Estimated Core Power (W)			Measured Full-System Power	Resource Utilization			
						Reported	Min	Max		LUTs	REGs	BRAMs	DSPs
Jin, 2014 [29]	Stereo	Kintex-6	640×480	60	15.6	10.6	1.93	2.67	none	61000	61000	165	0
Raj, 2014 [30]	DfD	Virtex-4	400×400	n/a	48.0	none	0.46	0.58	2W + camera	10028	7609	29	50
Mattoccia, 2015 [45]	Stereo	Spartan-7	640×480	32	9.2	none	0.44	0.68	2.5W	23749	9030	67	0
Ttofis, 2015 [65]	Stereo	Kintex-7	1280×720	64	55.3	2.8	0.92	1.53	none	57492	71192	302	458
Puglia, 2017 [55]	Stereo	Zynq Artix	1024×768	64	23.6	0.17	0.43	0.68	2W	29057	15208	80	0
Zhang, 2018 [69]	Stereo	Kintex-7	1920×1080	128	124.4	none	0.89	1.23	none	53190	40980	151	0
Lu, 2021 [40]	Stereo	Kintex-7	1024×480	128	57.0	none	1.39	2.30	none	50465	48046	125.5	8
Wang, 2022 [66]	Stereo	Zynq US+	1920×1080	256	344.2	3.35	2.03	2.20	2.8W + camera	92300	59300	332.5	0
Ours, on Kintex-7	DfD	Kintex-7	512×480	n/a	59.9	-	**0.42**	**0.55**	-	24173	21947	58	58
Ours, actual, on ECP5	DfD	ECP5	480×400	n/a	30.5	-	**0.24**	**0.31**	**0.6W**	47339†	35107†	95†	67†

Table S4. **Extended version of Tab. 1.** This table includes the reported power from other works. We re-did calculations from other works to account for differing pixel processing rates - our analysis favors methods that process pixels above 30 Mpix/s and penalize works below 30 Mpix/s. We under-estimate power for all works except [55].
† The internal design of the Lattice ECP5 means that its resource utilization values can't be directly compared to those of other FPGAs. Each LUT, BRAM, and DSP module on the ECP5 is smaller than on the 7-Series, so a design on the ECP5 will typically take more of these smaller resources.

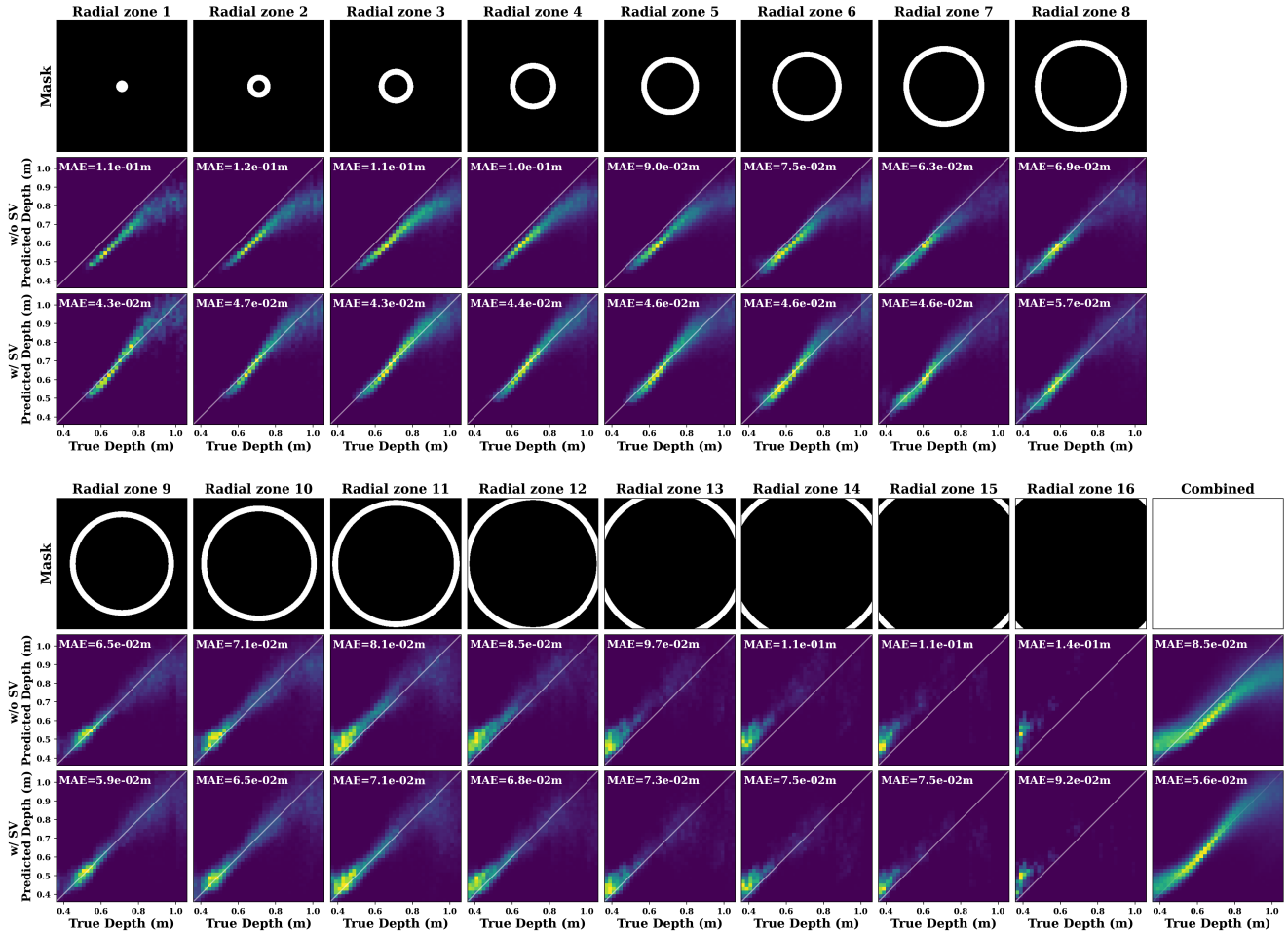

Figure S4. **Heatmap comparison of our method with and without field curvature correction.** We decompose an image in 16 radial regions (row 1) and show the predicted depth vs. true depth heatmaps (filtered with a 90% sparsity) for every radial regions and their combined region. We show that compared to using constant optical parameters (row 2), applying spatial variation (row 3) yields better depth predictions that are much more aligned with the diagonal, indicating better accuracy, in all radial zones. We also observe with the confidence filtering, radial zones in the center tend to perform better at longer depth ranges than the peripheral zones, which is well aligned with the observation of Petzval field curvature, where the focal planes are curved towards the lens.

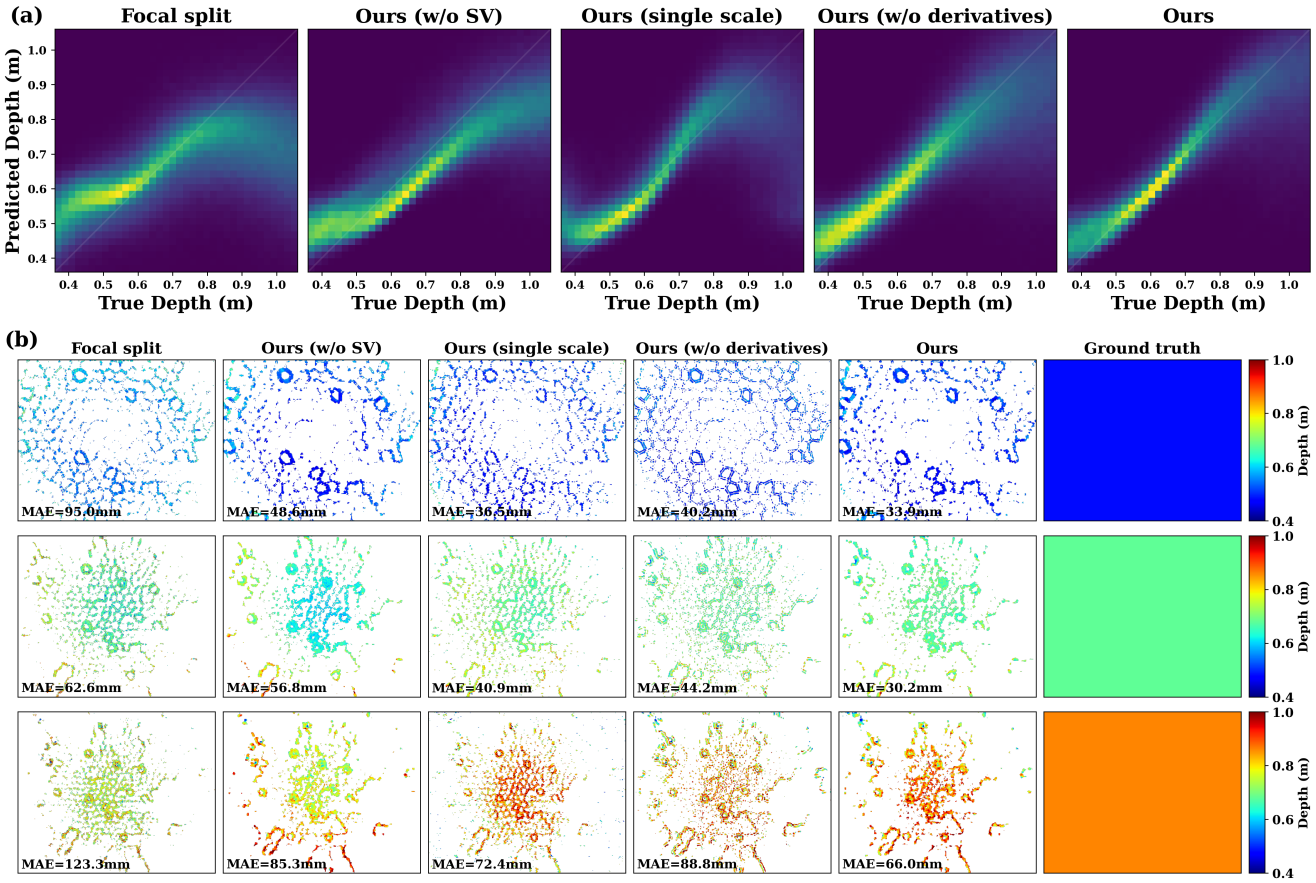

Figure S5. **Ablation studies.** (a) Heatmaps of Focal Split [42] and different variations of our method. (b) Predicted depth maps on the calibration dataset at three different depths from our front-parallel calibration dataset. We show that incorporating spatial variation, multiscale, and multi-derivative in our algorithm helps improve depth estimation in our lower power setting.

Component Description	Commercial off-the-shelf	Part number / path to design files on GitHub	Qty
FPGA development board	yes	LFE5UM5G-85F-EVN	1
Image sensor cables	yes	900-0150180325-ND	2
FT232H breakout board	yes	2264 (Adafruit)	1
16mm beamsplitter	yes	CCM5-BS016	1
M09 mounted lens	yes	AC050-008-A-ML	2
Lens adapter	yes	S05TM09	2
Image sensor boards	no	`pcb/ecp5-fpc-adapter`	2
Image sensor adapter	no	`pcb/hm0360-carrier`	1
FT232H adapter board	no	`pcb/ft232h-breakout`	1
3D printed optics fixture	no	`3d_printed_parts/dual_sensor_enclosure`	2

Table S5. A list of all of the significant parts making up our system. Commercial off-the-shelf components can be purchased at a store, while non-commercial off-the-shelf parts are those which we designed for this project and can be found in our GitHub repository (`https://github.com/NUBIVlab/SpiderCam`) using the path names. To replicate our work, our design files must be used to order these from manufacturers.

images in the 2 image sensors are offset. After assembling the system, we manually calculate a homography; at runtime, we apply it to both images to align them before processing as described in Sec. 3.2. To calculate the homography, we capture several photographs of a checkerboard pattern and manually align them using a custom tool found at `https://github.com/johnMamish/manual-homography-gui`. To capture the photographs for homography, the FPGA board must be reprogrammed with the Gateware image at `https:`

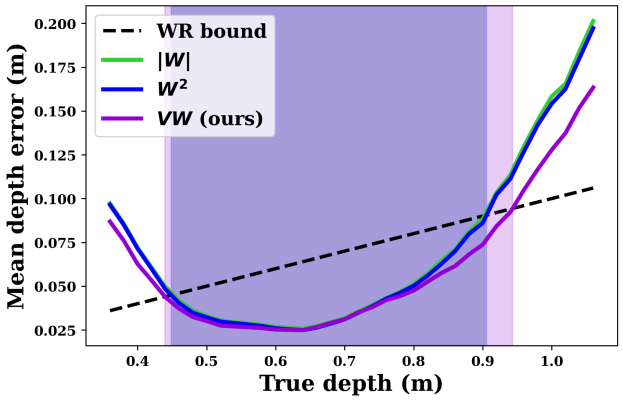

Figure S6. **Comparison of different confidence.** We show the mean depth error vs. true depth curves at 90% sparsity on the calibration data using different confidence metrics. We observed an increase in the working range with VW (numerator in Eq. (3)) as confidence compared with W^2 or $|W|$, which were used in previous work [41, 42].

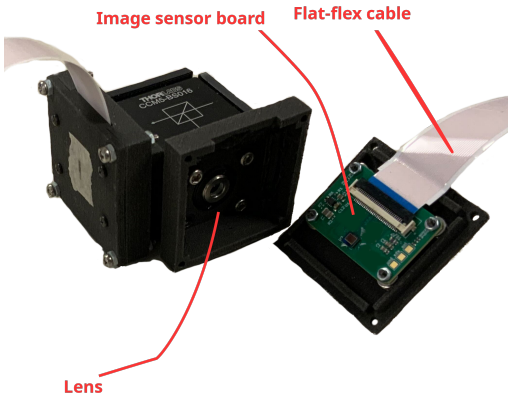

Figure S7. **Optical assembly.** This photograph shows the 3D printed mount that affixes the image sensor to the beamsplitter and aligns it to the lens. The image sensor, lens, and flat-flex cable connecting the optical assembly to the FPGA development board are visible.

//github.com/NUBIVlab/SpiderCam/tree/main/fpga/synth/camera_previewer.

S10. Depth Calibration

We calibrate our prototype with experimentally collected front parallel texture plans at known depths. We collect 56 pairs of images of the textured plane at a depth range of 0.24m to 1.36m, with a step size of 0.02 m. We calculate the mean absolute error (MAE) between the depth predictions of all image pairs and the true depth as the loss function for calibration. We only calculate the MAE loss on the 10% most confident pixels in each radial zone (see Sec. 3.6). We observed that by doing so, we prevent the optical constants from being biased by the noisy pixels and yield better depth predictions. We use the AdamW optimizer with a learning rate of 0.05 and optimize for 100 iterations. More details about our optimization parameters can be found in the configuration files within our GitHub repository. We save our optimized optical constants a, b, ω in Eqs. (1) and (3) in the supplementary numpy files.

During inference, we set constant confidence thresholds for each radial zone to ensure reliable depth predictions. We use a log uniform confidence threshold from 1×10^{-7} (radial zone 1) to 2×10^{-5} (radial zone 16). The peripheral radial zones have larger thresholds because they are more affected by field curvature and aberrations. The starting point and ending point of the thresholds are tuned to maximize the working range while maintaining a density of 5% within the working range.

In Fig. S4, we show heatmaps (with enforced 90% overall sparsity) in the 16 radial zones with and without spatial variation. We observe that when the optical constants are universal (row 2), the central and peripheral radial zones exhibit strong biases in the depth prediction, with the depth prediction being smaller in the central zones and larger in the periphery. This agrees with the fact that the focal plane is curved towards the lens at the periphery of the field of view due to Petzval field curvature [22]. When spatially-varying optical constants are used, depth predictions are concentrated on the diagonals and the bias is no longer visible, which indicates much higher accuracy. We also compare several design choices in our method with Focal Split [42] as an ablation study (all results are shown with an enforced 90% sparsity in all depth maps). In the heat map comparison in Fig. S5a, we show our depth predictions are much more concentrated on the diagonal compared to Focal Split, ours without spatial variation, ours with a single scale, and ours without derivatives. In Fig. S5b, we compare predicted depth maps on three image pairs at three different depths in our calibration dataset. We show that our method robustly outperforms others with better accuracy, especially in the challenging peripheral areas. These are strong evidences that the incorporation of spatial variation, dual-scale, and spatial derivatives is improving depth predictions. In Fig. S6, we compare three confidence metrics: $|W|$ used in previous DfDD methods, and W^2 and VW, which are available without extra computation due to the depth computation. We show that using VW (numerator in Eq. (3)) yields the widest working range with a fixed 90% sparsity (45.1 cm for $|W|$, 45.7 cm for W^2, 50.3 cm for VW).

S11. Additional scenes and video

See full display of the depth maps produced on device using the front-parallel textures and their respective MAE at

Table S6. **Full system power consumption** including image sensors, I/O, and FPGA, as reported in the literature and measured for several configurations of our method.

Work	Scales	Compute I_x, I_y	FPS	Overall power (mW)
[42]	-	-	2.1	4900
[55]	-	-	30	2000
[45]	-	-	> 30	< 2500
ours	2	yes	32.5	624
ours	2	no	32.5	562
ours	1	yes	32.5	489
ours	1	no	32.5	468
ours	2	yes	9	399
ours	2	no	9	387
ours	1	yes	9	356
ours	1	no	9	347

Fig. S8. This is the data that is concisely plotted as purple dots on Fig. 3. Furthermore, to show that the MAE from the font-parallel textures generalizes well, view Fig. S9 to see the MAE across additional real scenes. Generating the reference maps was done by placing object planes at known distances, then manually tracing the outlines and filling them in with appropriate values (or a gradient of values). Finally, we have the demo video titled 'demo.mp4' in the supplementary material, demonstrating real-time 32.5 FPS performance of our algorithm in action.

S12. Full System Power Evaluation

We measured the power consumption of our system in several configurations, shown in Tab. S6. To perform the power measurement, we used a Qoitech Otii Arc Pro DC energy analyzer. The Otii Arc Pro supplied our system with 4.2 V of power and recorded current consumption at 4 ksps. We used Otii's software to calculate average power consumption over a 5-second window. A representative power trace can be seen in figure Fig. S10.

During power measurements, the system calculated depth for real scenes similar to those in Fig. 2. To verify system functionality during measurement, we read data off as normal using a USB-to-Parallel converter and monitored data output. We designed the USB adapter board so that it cannot supply power to our system, but to rule out parasitic power supply via GPIO pins, we took power measurements with and without the USB adapter board attached and found that there was no noticeable difference.

S13. Power / Accuracy Trade-Off Details

In this section, we provide details on the power-accuracy tradeoff comparisons in Fig. 4 (see summary of working range, MAE, and core power in Tab. S7). We tested the power consumption and accuracy of several configurations of our system - by reducing the size of some computations, we can save power and reduce circuit size, but accuracy suffers, as can be seen in Tab. S7.

Table S7. **Power-accuracy tradeoff for alternate algorithms.** We report the working range, MAE, and core power in Tab. S7.

Scales	I_x, I_y	SV	WR (m)	MAE (m)	Core power (mW)
2	yes	yes	0.551	0.044	312
2	yes	no	0.311	0.058	280
2	no	yes	0.282	0.075	276
2	no	no	0.259	0.071	257
1	yes	yes	0.423	0.069	214
1	yes	no	0.210	0.076	195
1	no	yes	0.175	0.089	202
1	no	no	0.074	0.098	184

For all methods that use spatial variation (circles and stars), we start with the same log-uniform confidence threshold described in Sec. S10, and scale the thresholds together for the widest working range. For the other methods using a universal threshold (squares), we tune the threshold value for the widest working range. This is necessary because the number of estimates changes the overall range of possible confidence estimates. We calculate the working range and depth MAE over 0.4 m to 1.0 m using the pixels whose confidence is above the confidence threshold. In our code base, we provide an interface for users to change these options (spatial variation, number of scales, and spatial derivatives) in our method through a configuration file.

We characterized the power consumption of each configuration we tested as described in Sec. S8. We synthesized the SystemVerilog design for each different configuration and used vendor tools for the ECP5 to estimate the power consumption.

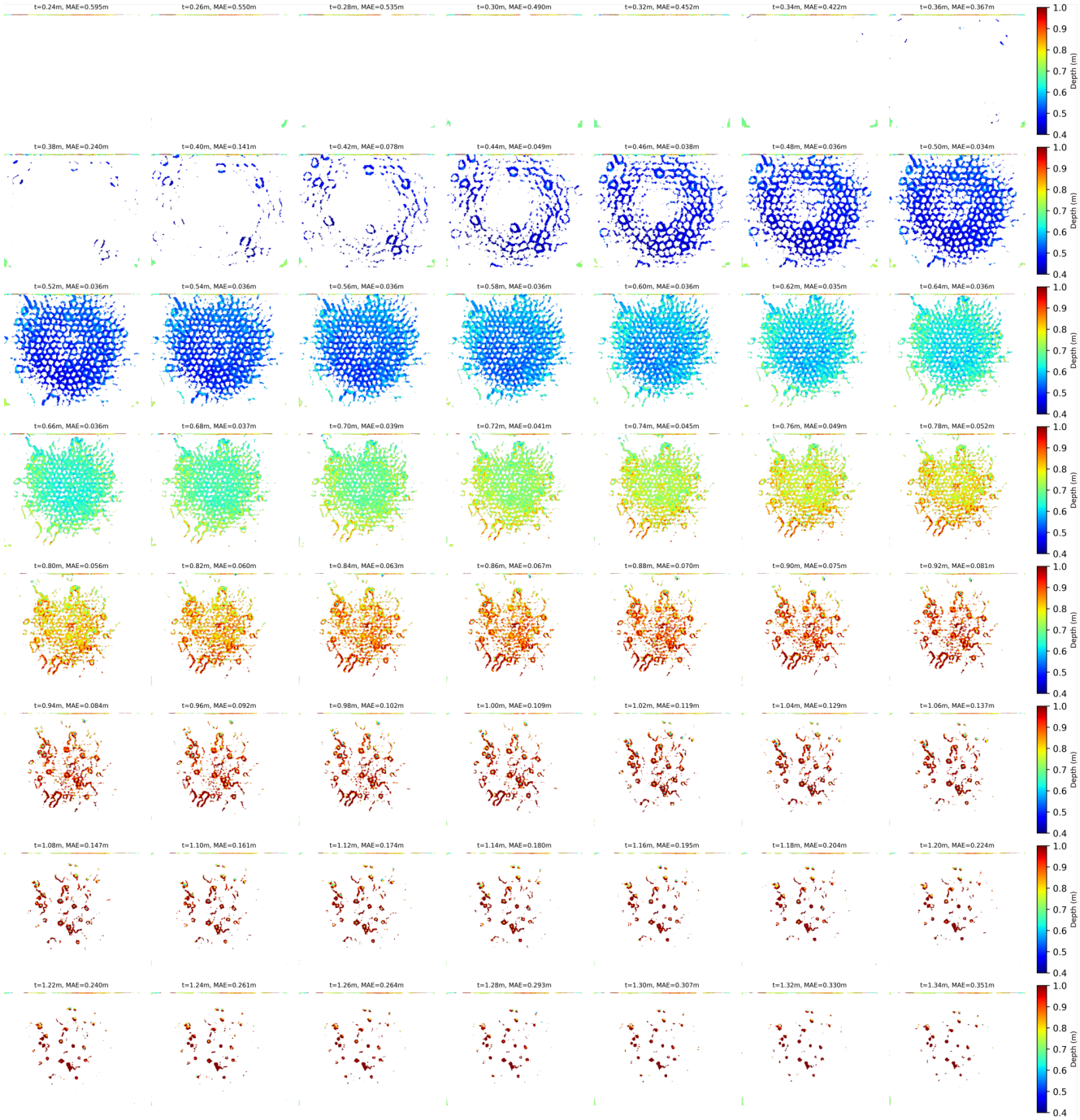

Figure S8. **Depth maps computed on-device for front-parallel textured plane.** This data was used to generate purple dots in Fig. 3. Note that density naturally falls off away from the center of the working range. Our confidence estimate captures edge textures with sufficient focus, and field curvature shifts the effective working range across the image. Edge effects dominate error in sparse images.

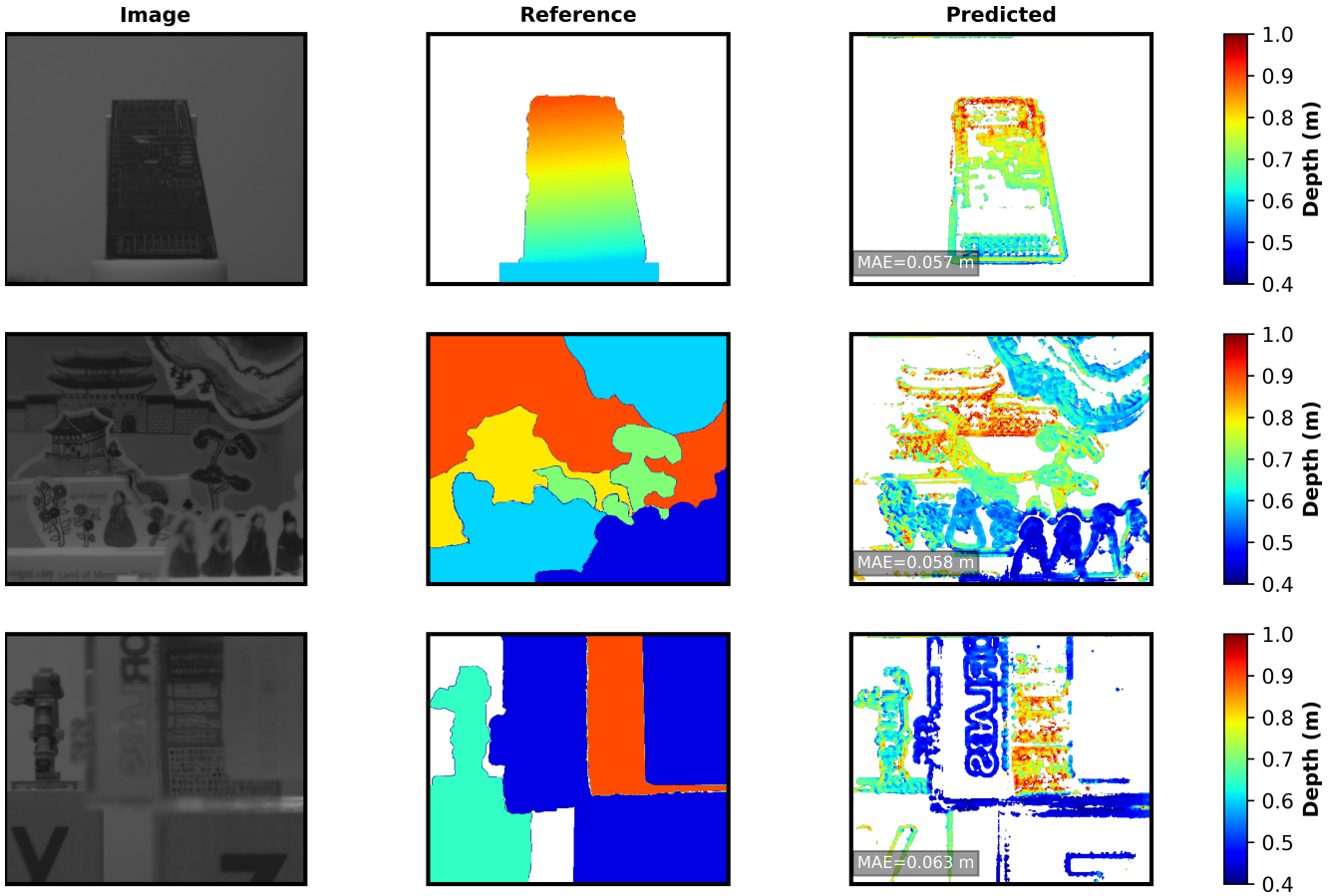

Figure S9. **Depth maps computed on-device with their references for real scenes.** Note MAE from top to bottom of 0.057 m, 0.058 m, and 0.063 m, consistent with data in Fig. S8

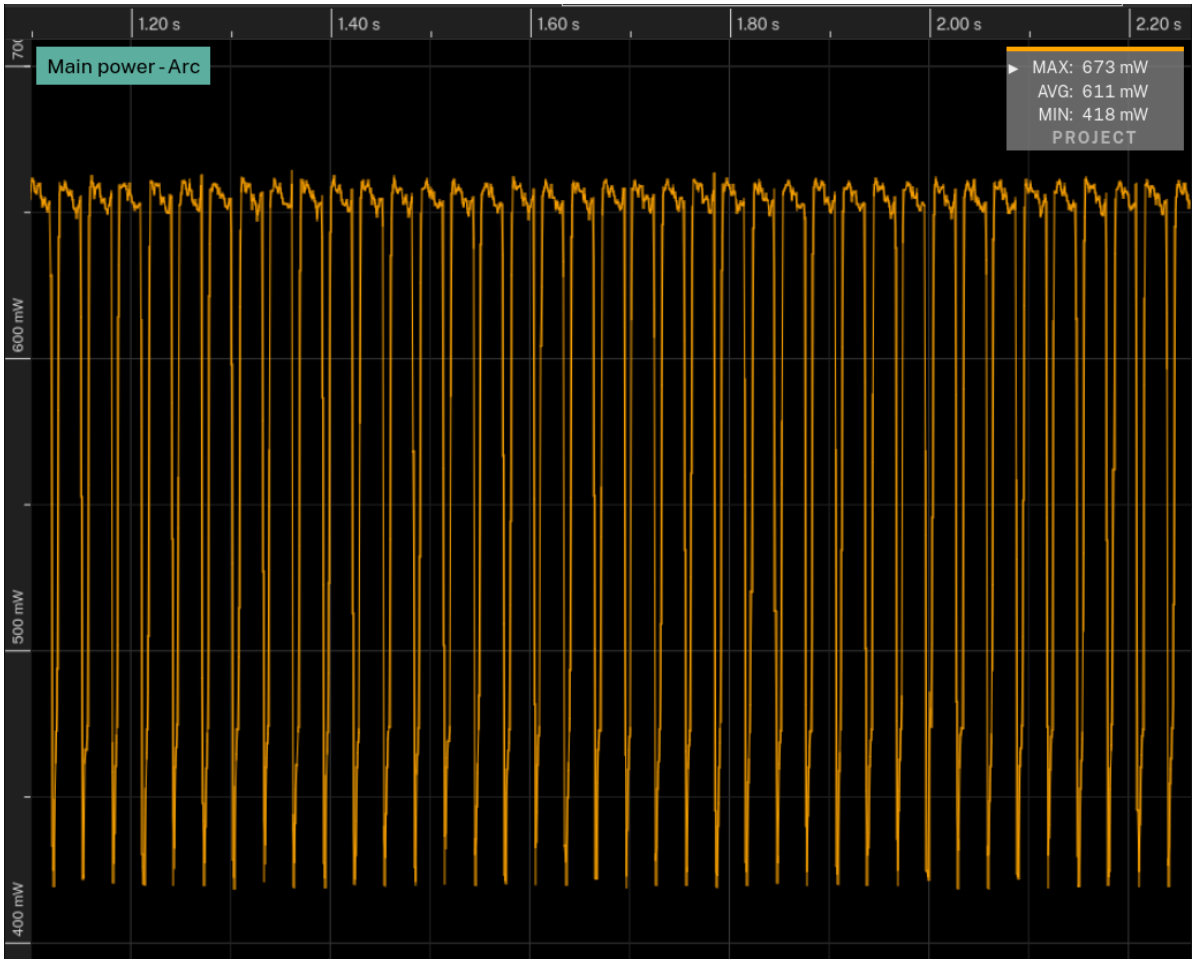

Figure S10. **Screenshot of representative power consumption trace.** We demonstrate how we used the Otii Arc Pro to measure power consumption for Tab. S6 by measuring power over time. This screenshot shows power for our system measured in a dual-scale configuration with dx and dy filters at 32.5 FPS. Notably, the frame rate is evident from the frequency of fluctuations in power consumption as seen in this graph - during active processing of a frame, power consumption is 650 mW; in-between frames, it drops to 420 mW. Our average power is 611 mW, which is slightly lower than our demonstrated 624 mW due to having captured this trace when the FPGA was cold compared to an extended period of usage during the recording of our demonstration that allowed the chip to warm up to its stable temperature. At the higher warm temperature, resistive losses in the device rise as well, resulting in slightly higher overall power draw.