electronics

MDPI

Article

Area-Efficient Vision-Based Feature Tracker for Autonomous Hovering of Unmanned Aerial Vehicle

Hyeon Kim [1], Jaechan Cho [1], Yongchul Jung [1], Seongjoo Lee [2] and Yunho Jung [1,*]

[1] School of Electronics and Information Engineering, Korea Aerospace University, Goyang-si 10540, Korea; hkim@kau.kr (H.K.); jccho@kau.kr (J.C.); ycjung@kau.kr (Y.J.)

[2] The Department of Information and Communication Engineering, Sejong University, Seoul 05006, Korea; seongjoo@sejong.ac.kr

* Correspondence: yjung@kau.ac.kr; Tel.: +82-2-300-0133

Received: 4 September 2020; Accepted: 25 September 2020; Published: 28 September 2020

Abstract: In this paper, we propose a vision-based feature tracker for the autonomous hovering of an unmanned aerial vehicle (UAV) and present an area-efficient hardware architecture for its integration into a flight control system-on-chip, which is essential for small UAVs. The proposed feature tracker is based on the Shi–Tomasi algorithm for feature detection and the pyramidal Lucas–Kanade (PLK) algorithm for feature tracking. By applying an efficient hardware structure that leverages the common computations between the Shi–Tomasi and PLK algorithms, the proposed feature tracker offers good tracking performance with fewer hardware resources than existing feature tracker implementations. To evaluate the tracking performance of the proposed feature tracker, we compared it with the GPS-based trajectories of a drone in various flight environments, such as lawn, asphalt, and sidewalk blocks. The proposed tracker exhibited an average accuracy of 0.039 in terms of normalized root-mean-square error (NRMSE). The proposed feature tracker was designed using the Verilog hardware description language and implemented on a field-programmable gate array (FPGA). The proposed feature tracker has 2744 slices, 25 DSPs, and 93 Kbit memory and can support the real-time processing at 417 FPS and an operating frequency of 130 MHz for 640 × 480 VGA images.

Keywords: feature tracker; flight control; hovering; system-on-chip; unmanned aerial vehicle (UAV)

1. Introduction

Unmanned aerial vehicles (UAVs), often called quadcopters or drones, have been widely applied to execute missions in various fields, such as precision agriculture, security, and surveillance [1]. One of the most popular and promising applications of UAVs is ground target surveillance that requires stable hovering of the UAV. Hovering is usually performed using a global positioning system (GPS) in an outdoor environment. However, the scope of application of hovering based only on GPS information is limited because GPS signals are often shadowed in dense environments, such as cities, and unavailable in indoor environments [2]. Therefore, a reliable technique for estimating the position of the UAV is required in GPS-denied environments. Inertial measurement units (IMUs), light detection and ranging (LiDAR), or camera sensors [3–10] are used for hovering in GPS-denied environments. Simultaneous localization and mapping (SLAM) techniques using depth sensors such as LiDAR or stereo-vision cameras build detailed maps of rooms and can thus be used for indoor hovering. However, they require very complex computations and have high power consumption. Among them, vision-based techniques with camera sensors are particularly suitable for UAVs because of their lower power consumption and weight compared with other techniques [5,6]. Vision-based hovering is a method to track the motion of a UAV using image sequences acquired from a downward-facing

camera and then compensate for the current motion of the UAV based on the tracked motion. Therefore, accurate motion tracking of the UAV is essential.

Several algorithms for tracking the movement of UAVs have been studied [5–10]. The template matching algorithm was used in [7], which estimates the deviation by calculating the correlation between the features of sequential frames using the sum of absolute differences (SAD) or the sum of squared differences (SSD). Even though this algorithm has low computational complexity, its tracking accuracy is very low because it determines whether the templates match only via the values of the SSD or the SAD. The Kanade–Lucas–Tomasi (KLT) algorithm can track the movement of the UAV by detecting the features in an image and estimating the optical flow for each feature [8]. Compared with the template matching method, the KLT algorithm has higher accuracy because all computations are kept at a subpixel accuracy level using bilinear interpolation [10].

The KLT algorithm consists of two processes: feature extraction using the Shi-Tomasi algorithm and feature tracking using Lucas–Kanade (LK) optical flow estimation (OFE) [11]. The Shi–Tomasi algorithm defines features based on the eigenvalues of the Hessian matrix [12] and exhibits excellent performance in feature tracking applications. However, the computation of eigenvalues is computationally intensive and becomes a bottleneck for real-time vision tasks, such as high-frame-rate video processing. The LK-OFE algorithm finds the flow vector of features by minimizing the sum of squared error between two images. It exhibits excellent performance and has been used in many applications [13–18]. However, there is a tradeoff between local accuracy and robustness when choosing the window size. A small window is preferable to avoid smoothing out the details of the images, but a large window is required to handle large motion. To overcome this problem, the pyramidal LK (PLK) algorithm was proposed in [19]. The PLK algorithm can track large motions even with a small search window by applying iterative processing to each level of an image pyramid. While the PLK overcomes this tradeoff, it still has a problem in that iterations increase computational complexity.

In this paper, we propose a feature tracker based on the Shi–Tomasi and PLK algorithms and present an area-efficient hardware architecture and FPGA-based implementation results. To reduce complexity, we simplified complex equations while maintaining performance. In addition, we developed a structure that shares common computations between the Shi–Tomasi and PLK algorithms. The rest of this paper is organized as follows. Background for the Shi–Tomasi and PLK algorithms is presented in Section 2. In Section 3, we describe a hardware architecture for the proposed feature tracker. Section 4 presents the results of our implementation and verification on an FPGA, and experimental results from real flight environments are presented in Section 5. Finally, we conclude the paper in Section 6.

2. KLT Feature Tracking Algorithm

2.1. Shi–Tomasi Algorithm

The Shi–Tomasi algorithm for feature detection was proposed in [12]. It is based on the 2×2 Hessian matrix $\mathbf{H}$:

$$\mathbf{H} = \begin{bmatrix} g_y^2 & g_y g_x \\ g_y g_x & g_y^2 \end{bmatrix}, \tag{1}$$

where g_x and g_y are gradients in the horizontal and vertical directions, respectively. The eigenvalues of $\mathbf{H}$ indicate the type of intensity change within the window centered at a given point. When both eigenvalues are small, the point is in a flat region. When one is large and the other is small, the point is at an edge. If both eigenvalues are large, the point is considered a corner [12]. To detect corners that are favorable for tracking, the minor eigenvalue is compared with a predefined threshold T as follows:

$$\min(\lambda_1, \lambda_2) > T, \tag{2}$$

where λ_1 and λ_2 are the eigenvalues. When a pixel satisfies Equation (2), it is considered a candidate for a good feature to track. To obtain non-overlapping features, all overlapped features within an $n \times n$ window centered at each candidate feature are deleted [11,12].

2.2. Pyramidal LK Optical Flow Estimation

Assuming that the brightness constancy constraint is satisfied, the LK-OFE algorithm computes the flow vectors of features between the current frame and the previous frame in image sequences [11]. The brightness constancy constraint can be expressed as follows:

$$I(x, y, t + \tau) = I(x - \delta x, y - \delta y, t), \tag{3}$$

where $I(x, y, t + \tau)$ is the intensity of the pixel corresponding to the feature point at $\mathbf{x}= (x, y)$ in the $(t + \tau)$-th frame, and $I(x - \delta x, y - \delta y, t)$ is the intensity of the pixel shifted by δx and δy in the x and y directions, respectively, in the t-th frame, which is the previous frame. The amount of motion $\mathbf{d} = (\delta x, \delta y)$ is called the displacement vector of point $\mathbf{x}$. If we re-define $J(\mathbf{x}) = I(x, y, t + \tau)$ and $I(\mathbf{x} - \mathbf{d}) = I(x - \delta x, y - \delta y, t)$, the value of d that minimizes the residue error ε can be calculated as follows:

$$\varepsilon = \sum_{j \in W} [I(\mathbf{x}_j - \mathbf{d}) - J(\mathbf{x}_j)]^2 w_j, \tag{4}$$

where w_j is a Gaussian kernel and W is a small window centered at one of the feature points. When $\mathbf{d}$ is small, the intensity function can be approximated by its Taylor series as follows:

$$I(\mathbf{x} - \mathbf{d}) = I(\mathbf{x}) - \mathbf{g} \cdot \mathbf{d}, \tag{5}$$

where $\mathbf{g}$ is the gradient vector $\left(\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\right)$ at $\mathbf{x}$. Then, we can rewrite the residual error as

$$\varepsilon = \sum_{j \in W} [h(\mathbf{x}_j) - \mathbf{g} \cdot \mathbf{d}]^2 w_j, \tag{6}$$

where $h(\mathbf{x})$ denotes $I(\mathbf{x}) - J(\mathbf{x})$. Because the residue is a quadratic function of the displacement $\mathbf{d}$, the minimization can be done in closed form as

$$\sum_{j \in W} (h(\mathbf{x}_j) - \mathbf{g} \cdot \mathbf{d}) \mathbf{g} \cdot w_j = 0. \tag{7}$$

Because $(\mathbf{g} \cdot \mathbf{d})\, \mathbf{g} = \left(\mathbf{g}\mathbf{g}^T\right) \mathbf{d}$ and $\mathbf{d}$ is assumed to be constant within W, we have

$$\mathbf{G}\mathbf{d} = \mathbf{e}, \tag{8}$$

where the $\mathbf{G}$ is symmetric 2×2 matrix

$$\mathbf{G} = \sum_{j \in W} \mathbf{g}\mathbf{g}^T w_j = \begin{bmatrix} \sum\limits_{j \in W} w_j g_y^2 & \sum\limits_{j \in W} w_j g_y g_x \\ \sum\limits_{j \in W} w_j g_y g_x & \sum\limits_{j \in W} w_j g_x^2 \end{bmatrix}, \tag{9}$$

and $\mathbf{e}$ is a two-dimensional vector

$$\mathbf{e} = \sum_{j \in W} h(\mathbf{x}_j) \mathbf{g} w_j = \begin{bmatrix} \sum\limits_{j \in W} w_j g_y h(\mathbf{x}_j) \\ \sum\limits_{j \in W} w_j g_x h(\mathbf{x}_j) \end{bmatrix}. \tag{10}$$

However, the result of Equation (8) usually contains some errors. Therefore, **d** is determined via iterations. The coordinates of the current feature are compensated by **d** from the previous step, and images are resampled via bilinear interpolation to achieve subpixel accuracy.

To overcome the tradeoff between accuracy and robustness in LK-OFE, the PLK algorithm was proposed in [19]. Let I_t^L be the image corresponding to pyramidal level L for the t-th image, where $L = 0, 1, 2 \ldots, L_m$ are generic pyramidal levels. I_t^0 is essentially the highest resolution image (the raw image). The PLK algorithm sequentially tracks from the highest-level image $I_t^{L_m}$ to the lowest-level image I_t^0. The pyramidal image I_t^L is obtained via decimation filtering from I_t^{L-1} as follows:

$$\begin{array}{l} I^L(x,y) = \frac{1}{4} I^{L-1}(2x,2y) + \\ \frac{1}{8}\left\{ I^{L-1}(2x-1,2y) + I^{L-1}(2x+1,2y) + I^{L-1}(2x,2y-1) + I^{L-1}(2x,2y+1) \right\} + \\ \frac{1}{16}\left\{ I^{L-1}(2x-1,2y-1) + I^{L-1}(2x+1,2y+1) + I^{L-1}(2x-1,2y+1) + I^{L-1}(2x+1,2y-1) \right\}. \end{array} \tag{11}$$

Figure 1 depicts the overall procedure of the PLK algorithm for finding the flow vector. $\mathbf{d}_t^L$ denotes the displacement at level L. $\mathbf{v}_t^L$ is the initial guess that propagates from level L and is calculated using $\mathbf{d}_t^L$ and $\mathbf{v}_t^{L+1}$ as follows:

$$\mathbf{v}_t^L = \begin{cases} 2\mathbf{d}_t^{L_m} & , \quad L = L_m \\ 2(\mathbf{v}_t^{L+1} + \mathbf{d}_t^L) & , \quad 0 < L < L_m, \\ \mathbf{v}_t^1 + \mathbf{d}_t^0 & , \quad L = 0 \end{cases} \tag{12}$$

where $\mathbf{v}_t^L$ is propagated to the lower level by being used as a new initial guess at level $L-1$. This propagation is repeated down to level 0, and thus the flow vector $\mathbf{v}_t^0$ of the t-th image is finally obtained.

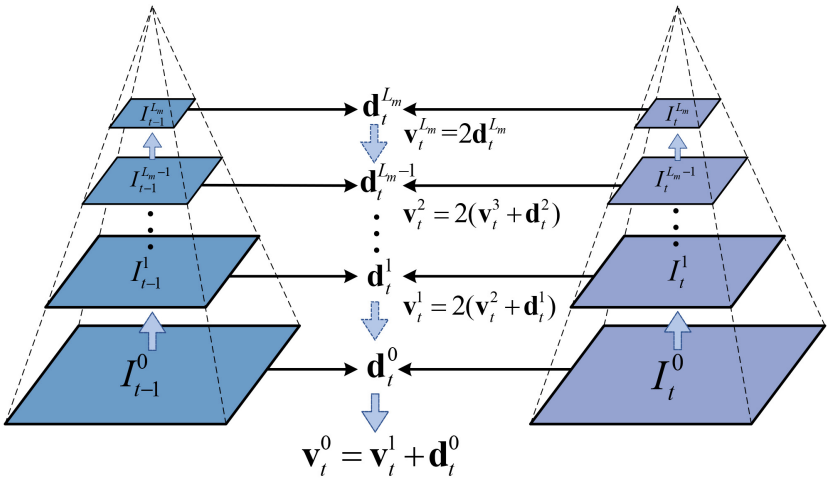

Figure 1. Procedure of the pyramidal Lucas–Kanade (PLK) algorithm.

3. Hardware Architecture of Proposed Feature Tracker

Figure 2 shows the overall architecture of the proposed feature tracker. The proposed feature tracker detects the features from the first frame I_0, and then tracks the features from the next frames I_1, I_2, I_3, …, I_{N-1}, where N is the frame number. The calculated motion vector $(u, v)^T$ for each frame is transferred to the UAV's flight controller to perform hovering in real time. After tracking N frames, feature detection is performed again at frame I_N, and then feature tracking is performed iteratively for subsequent frames, I_{N+1}, I_{N+2}, …, I_{2N-1}.

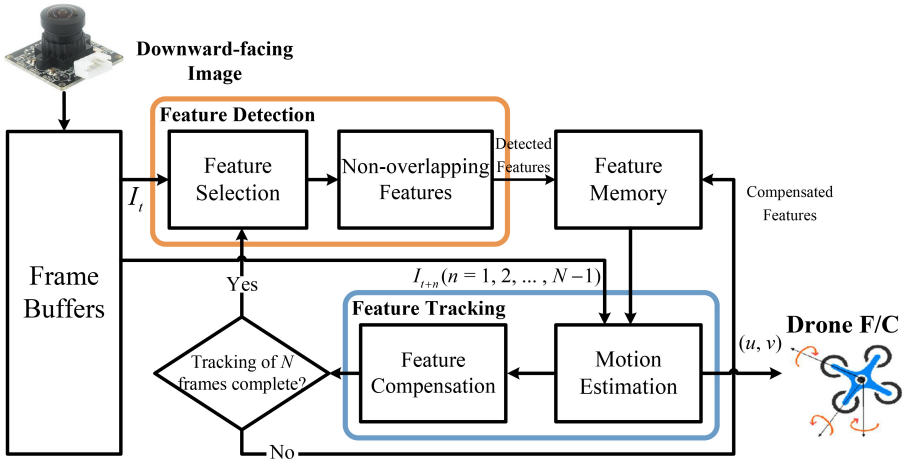

Figure 2. Overall architecture of the proposed feature tracker.

Figure 3 shows a block diagram of the proposed feature tracker, which consists of a Hessian-matrix calculation unit (HCU), a feature detection unit (FDU), and a feature tracking unit (FTU). Because the Shi–Tomasi and PLK algorithms involve the same calculation of gradients and multiplications, the FDU and the FTU share the HCU to reduce hardware complexity. In addition, the HCU, FDU, and FTU are all designed with a pipelined structure for high-speed operation and lower memory usage.

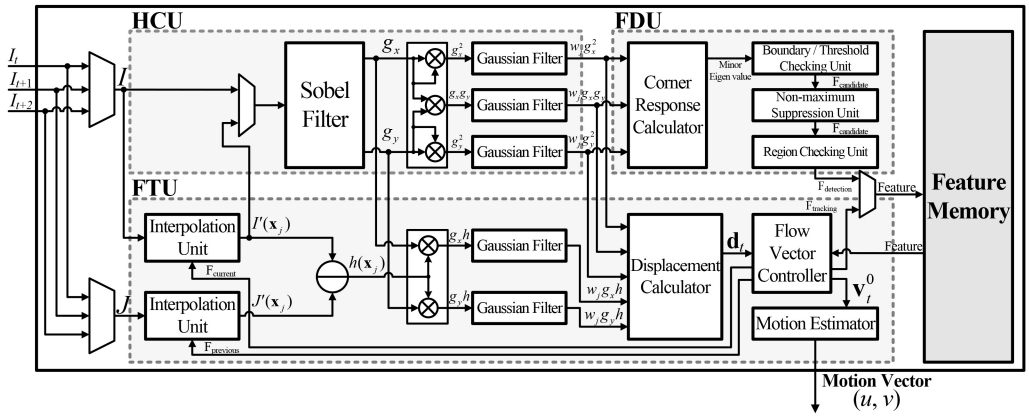

Figure 3. Block diagram of the proposed feature tracker.

3.1. Hessian-Matrix Calculation Unit (HCU)

The HCU shown in Figure 3 consists of a Sobel filter and Gaussian filters. The HCU calculates matrix **H** in Equation (1) for the Shi–Tomasi algorithm and matrix **G** in Equation (9) for the PLK algorithm. The Sobel filter calculates the gradients in the horizontal and vertical directions, and the Gaussian filters smoothen these gradients using a Gaussian kernel. Gaussian and Sobel filters are implemented with a line buffer structure to carry out the convolution operation. The line buffer unit (LBU), which is shown in Figure 4, converts sequential input data to the form of an $n \times n$ window. The LBU can be implemented without additional frame buffers because it reuses the input data line by line. The five Gaussian filters and the Sobel filter in the HCU, the two interpolation units in the FTU, and the NMS (non-maximum suppression) unit in the FDU were also implemented using LBUs. Consequently, the memory size was reduced from $9R_x \times R_y$ to $18R_x$, where R_x and R_y denote the horizontal and vertical resolutions of the input image, respectively.

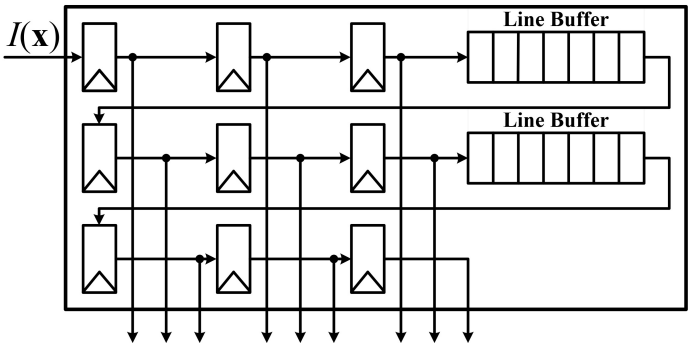

Figure 4. Block diagram of line buffer unit (LBU).

3.2. Feature Detection Unit (FDU)

The FDU is composed of a corner-response calculation unit (CRCU), a boundary and threshold checking unit (BTCU), an NMS unit (NMSU), and a region checking unit (RCU). The CRCU determines the minor eigenvalue, which is a corner response. If the elements of the matrix **H** in Equation (1) are p, r, r, and q (in that order), the characteristic equation of **H** can be expressed as

$$\lambda^2 - (p+q)\lambda + pq - r^2 = 0. \tag{13}$$

By applying the quadratic formula to Equation (13), the minor eigenvalue is simply computed as

$$\lambda_{\min} = \frac{(p+q) - \sqrt{(p-q)^2 + 4r^2}}{2}. \tag{14}$$

The BTCU selects points that have corner probabilities higher than a predefined threshold as feature candidates. In addition, the BTCU excludes feature candidates located at the boundary because these may be left out of the image during the tracking process.

After feature candidates are selected by the BTCU, the final features are selected by the NMSU and RCU. Because points close to good features typically have high corner responses, selected candidates can overlap. To avoid tracking essentially the same point multiple times, the NMSU and RCU enforce a minimum distance between features. The NMS technique picks only one feature with the best corner response in a window and then removes the remaining features that overlap with it and iterates [11]. Because iterating causes latency to increase, the NMSU is designed with a hardware-friendly structure using an LBU. As depicted in Figure 5a, the NMSU is composed of LBU and comparison units, which are shown in Figure 5b. Instead of carrying out a sorting process, NMS is performed by removing feature candidates when the input corner response is not the largest in the window. Subsequently, the RCU divides the image into k regions and selects the first candidate from the NMSU in each region as the final feature.

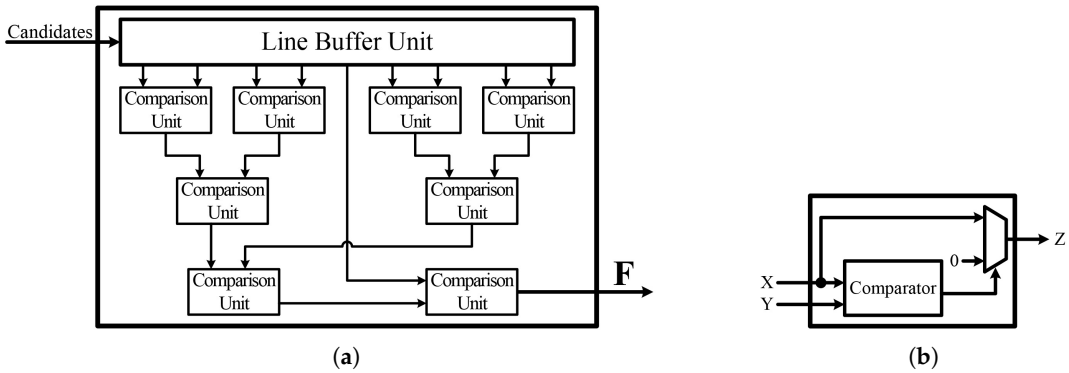

Figure 5. Block diagrams of (**a**) the non-maximum suppression unit (NMSU) and (**b**) a comparison unit.

3.3. Feature Tracking Unit (FTU)

The FTU consists of interpolation units, Gaussian filters, a displacement calculator, a flow vector controller, and a motion estimator, as shown in Figure 3. The traditional PLK algorithm starts tracking from the highest-level image I^{L_m} to the lowest-level image I^0. However, because the pyramidal images are constructed in order from the lowest-level image I^0 to the highest-level image I^{L_m}, performing decimation filtering to lower-level images as in Equation (11) requires additional memory to store the generated pyramidal images. Therefore, we use simple down-sampling instead of decimation filtering. Simple down-sampling enables the construction of an image pyramid without incurring additional processing time and memory by controlling the read address of the frame buffer according to the pyramidal level. Through experiments, we found that the degradation of the tracking performance when using down-sampling instead of decimation filtering is less than the root-mean-square error (RMSE) of one pixel.

The construction of pyramidal images via down-sampling is carried out in the interpolation unit. Figure 6 shows a block diagram of the interpolation unit, which consists of two blocks: a window region extractor (WRE) and a bilinear interpolator. The WRE operates as a read controller of frame buffers for the window centered on the feature corresponding to the current pyramidal level. The bilinear interpolator calculates the intensity value corresponding to the coordinate expressed as a real number via bilinear interpolation.

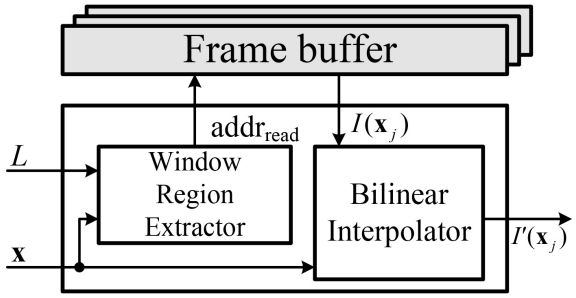

Figure 6. Block diagram of the interpolation unit.

Figure 7 depicts the WRE, which calculates the read address of the frame buffer for down-sampling. If pixel data are stored in the frame buffer in the horizontal direction, the read address of the frame buffer is calculated as

$$\text{addr}_{\text{read}}(x, y, L) = 2^L(x + c_1 - w_s - 1) + 2^L(y + c_2 - w_s - 1)R_y, \tag{15}$$

and the address of each pyramidal level is connected to the multiplexer. The final read address is selected based on the current pyramidal level. x and y are the coordinates of the feature, c_1 and c_2 denote the row and column counter values, w_s is the size of the window, R_y is the vertical resolution, and L is the pyramidal level.

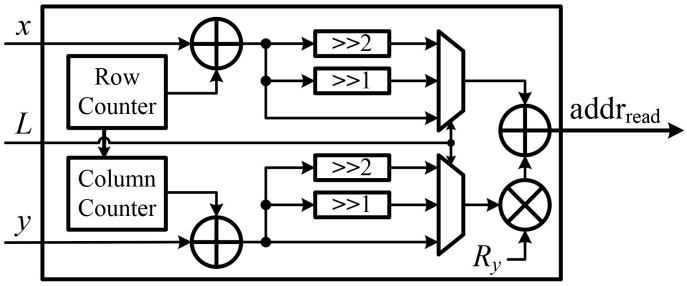

Figure 7. Block diagram of the window region extractor.

The coordinates of the feature compensated by the calculated displacement are real numbers, not integers. Therefore, bilinear interpolation is required to determine the intensity of the pixel corresponding to the real number coordinates and achieve subpixel accuracy. Bilinear interpolation involves two linear interpolations, one in the horizontal direction and one in the vertical direction. Because a linear interpolation can be performed via a convolution operation with kernels determined by the coordinates of the features, the bilinear interpolator was designed using the LBU as shown in Figure 8.

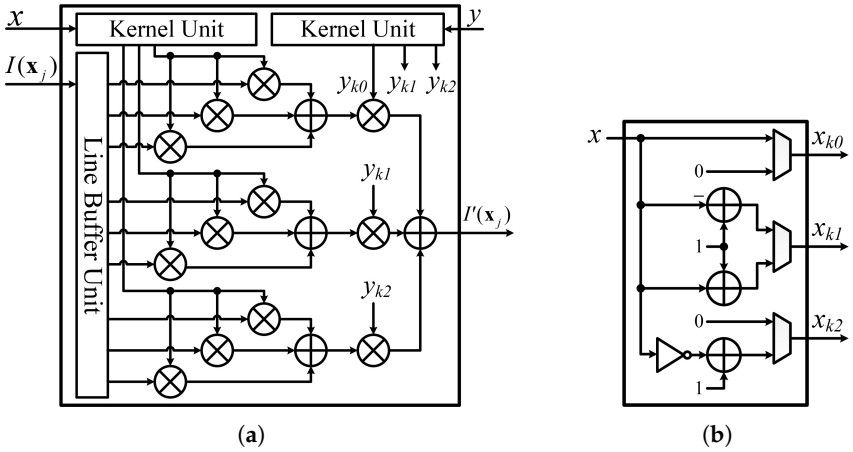

Figure 8. Block diagram of (**a**) the bilinear interpolator and (**b**) a kernel unit.

After interpolation, the values of w_j, $\mathbf{g}$, and h are accumulated to obtain the elements of Equations (9) and (10). The displacement calculator then calculates $\mathbf{d}$ as the product of a vector and an inverse matrix, as shown in Figure 9.

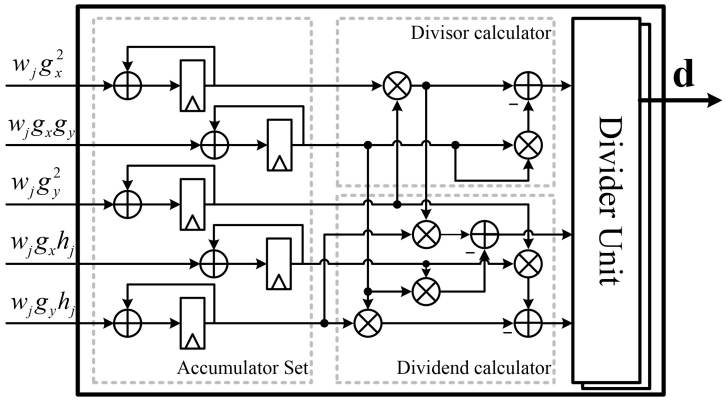

Figure 9. Block diagram of the displacement calculator.

To calculate the 2×2 inverse matrix, division by the determinant of $\mathbf{G}$ is required. Because the division operation is highly complex, we use an efficient divider unit considering the characteristics of the proposed feature tracker, as shown in the flowchart in Figure 10. Here, z is the dividend, d is the divisor, q_{int} is the integer part of the quotient, q_{dec} is the decimal part of the quotient, and f is the number of decimal places. The latency of this divider unit is proportional to the size of the quotient. However, because the value of each element of the displacement vector is less than n when the size of the search window is $n \times n$, the latency of the divider unit is within $(q_{int} + f)$ clock cycles, which is approximately 10 clock cycles. Therefore, the divider unit can be implemented with approximately 35% less area than the general divider unit in [20] using shifters and an adder without causing performance degradation.

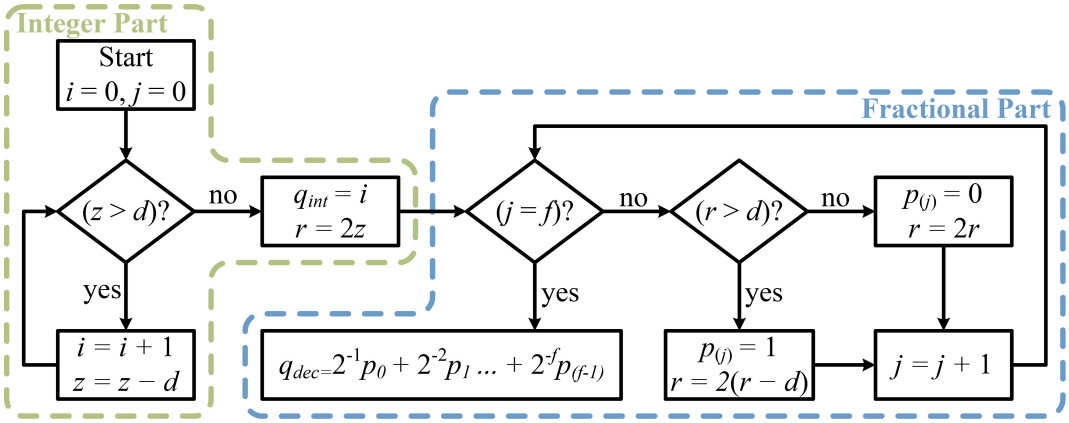

Figure 10. Flowchart of the divider unit.

After the calculation of the displacements, the feature memory is iteratively updated via the summation of the displacement and the stored coordinates of features. The final motion vector of a feature is then calculated by performing a pyramidal propagation from level L to level 0. After the tracking process for all features is performed, the tracking process is repeated for the next frame.

To calculate $(u, v)^T$, which is the motion vector of a single value, from several flow vectors, a motion estimator based on a histogram is used. The motion estimator analyzes the histogram of the flow vectors. The histogram is generated by dividing the entire range of the flow vectors into a series of intervals and then counting how many vectors fall into each interval [21].

4. FPGA Implementation Results

The proposed feature tracker was designed using Verilog, a hardware description language, and implemented on a Xilinx Virtex7 XC7VX330T FPGA device [22]. Thus, the proposed feature tracker was implemented with 2744 slices, 7459 slice LUTs, 25 DSPs, and a 93 Kbit block RAM. Its power consumption is 108 mW at a maximum operation frequency of 130 MHz, as shown in Table 1.

Table 1. Implementation results of the proposed feature tracker.

Block	Slices	Slice LUTs	DSP48s	Block RAM	Power Consumption
HCU	775	1937	3	34 Kb	32 mW
FDU	660	1633	2	21 Kb	24 mW
FTU	1309	3859	20	38 Kb	52 mW
Total	2744	7459	25	93 Kb	108 mW

A comparison of the FPGA resources required for the proposed feature tracker and those of previous works [23–25] is presented in Table 2. The tracker proposed in [25] for embedded vision applications can extract and track features at very high speeds by making the inter-frame displacement very small. This approach significantly reduces the time required to extract all features and exhibits comparable performance with the proposed design in terms of the number of maximum features and normalized execution time. However, compared with [25], the proposed feature tracker can reduce the maximum number of slices by 56%, slice LUTs by 65%, and block RAM by 97%. This is because the HCU is fully shared, several computations (such as division and minor eigen value calculation) are simplified, and the pipeline structure was designed without additional block RAM. Even though the design proposed in [23] requires fewer hardware resources, it was implemented without the pyramidal scheme.

Table 2. Comparison of the proposed tracker and previous works.

Target FPGA	Circuit	Slices	Slice LUTs	Block RAM
Virtex2	[23]	550	1237	90 Kb
Virtex5	[24]	5702	24,002	1763 Kb
	[25]	7334	29,376	3991 Kb
	Proposed	3203	10,324	93 Kb
Virtex7	Proposed	2744	7459	93 Kb

Table 3 shows the comparison results in terms of processing speed. To make a fair comparison, we normalized the execution time as time per pixel at a clock frequency of 60 MHz. As shown in Table 3, the execution time per pixel and feature of the proposed feature tracker is lowest among the different trackers considered owing to its pipelined structure and pyramidal image construction with no latency. In addition, the proposed tracker can track more features than those of previous works at the same resolution. Moreover, the execution time per feature of the proposed tracker is also lowest. Therefore, the proposed tracker is well suited for autonomous hovering of UAVs because it can support a good tracking performance with low hardware complexity and high processing speed, as evidenced in Tables 2 and 3.

Table 3. Comparison of the processing speed of our approach with previous works.

	[23]	[24]	[25]	Proposed
# of iterations	N.A.	4	3	4
Height of pyramid	1	3	1	3
Frequency (MHz)	125	209	182	130
Resolution	320 × 240	720 × 480	1024 × 768	640 × 480
Execution time per frame @60 MHz (s)	0.0205	N.A.	0.0135	0.0051
Execution time per pixel (ns)	266.67	N.A.	17.17	16.88
Maximum features	49	100	1000	171@(720×480) 1554@(1024×768)
Execution time per feature @1024×768 (ns)	4280	N.A.	13.5	8.5

5. Experiment Results

To evaluate the performance of the proposed feature tracker, we used the DJI Phantom 4 [26]. Performance evaluations were performed by comparing the trajectory of motion vectors obtained from the proposed feature tracker based on the images from the downward-facing camera with the movement trajectory derived from GPS coordinates information from the DJI Phantom 4.

Because the unit of the motion vector of the proposed feature tracker is pixels and that of the GPS-based movement trajectory is meters, it is necessary to convert the former to meters as follows:

$$x_m = \frac{Hu}{f_L} \text{ and } y_m = \frac{Hv}{f_L}, \tag{16}$$

where H is the altitude of the drone and f_L is the focal length, which is the distance from the camera's projection center to the image sensor [27]. Subsequently, the movement trajectory of the UAV was calculated based on the histogram of the motion vectors expressed in meters. The red boxes indicate the position of the features in the previous frame, and the yellow lines indicate the flow vectors. The green arrow indicates a representative vector calculated from the histogram of the flow vectors. When the motion vector is (−0A, FE) in hexadecimal, we can derive that the UAV is moving at a speed of 2.13 cm per frame using Equation (16), as shown in Figure 11.

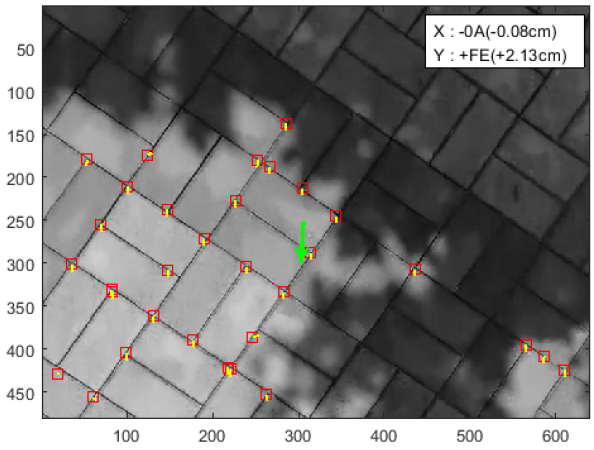

Figure 11. Motion vector in metric units.

The experiment was conducted using images obtained by flying for approximately 10 s at an altitude of 2 m on lawn, asphalt, and sidewalk block environments. For comparison with GPS-based trajectories, we unavoidably conducted experiment in outdoor environments. Table 4 summarizes the verification environments.

Table 4. Summary of verification environments.

Drone	DJI Phantom 4
Environments	Lawn/Asphalt/Sidewalk blocks
Moving distances	8.3 m/12 m/6.8 m
Altitude	2 m
Flight time	10 s
Resolution	640 × 480
FPS	30
Field of View	94°
Focal length	20 mm

Figure 12 shows the trajectories obtained by accumulating the metric moving distance of the proposed feature tracker and the movement trajectory of the Phantom 4 obtained based on the GPS coordinates. The proposed design tracks a moving path similar to the GPS-based results in each environment. Table 5 shows the RMSE normalized by moving distance of the results obtained using the proposed feature tracker and a GPS-based approach. The proposed feature tracker exhibited a similar performance with an average normalized RMSE of 0.039. Therefore, the proposed feature tracker should be able to support the hovering of UAVs even in GPS-denied environments, such as indoor environments.

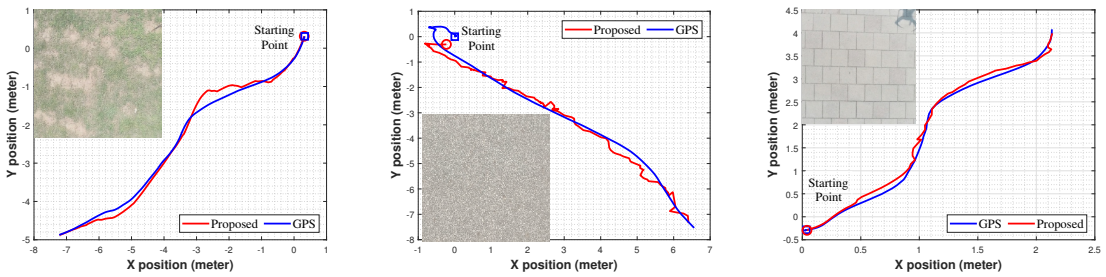

Figure 12. Comparison of the proposed feature tracking results with GPS-based results on (**a**) lawn, (**b**) asphalt, and (**c**) sidewalk blocks.

Table 5. Normalized root-mean-square error (NRMSE) of the proposed feature tracker.

Environment	NRMSE
Lawn	0.044
Asphalt	0.045
Sidewalk block	0.029

6. Conclusions

In this paper, we proposed an area-efficient hardware architecture to perform feature tracking to support the autonomous hovering of UAVs. We present a fast and area-optimized hardware design based on a robust and accurate pyramidal KLT algorithm. To reduce the area of the feature tracker, we leveraged the fact that the PLK and Shi–Tomasi algorithms share certain computations. The feature tracker was implemented with 2744 slices, 7459 slice LUTs, 25 DSPs, and 93 Kbit of on-chip memory on a Xilinx Virtex7 XC7VX330T device, which represents a reduction in the number of maximum slices of 56%, slice LUTs of 65%, and block RAM of 97% compared with existing feature tracker implementations. We experimented in various flight environments to verify our implementation, and we confirmed that the proposed feature tracker consistently tracks the motion of the drone with GPS-based trajectories. The proposed feature tracker supports real-time processing at 417 FPS and an operating frequency of 130 MHz for VGA images (640 $\times$ 480).

Author Contributions: H.K. designed the algorithm, performed the hardware architecture design and simulation and wrote the paper. J.C. conducted the hardware design and experiments. Y.J. (Youngchul Jung) and S.L. performed the implementation on FPGA and revised the manuscript. Y.J. (Yunho Jung) conceived and led the research, analyzed the experimental results, and wrote the paper. All authors have read and agree to the published version of the manuscript.

Funding: This work was supported by the Technology Innovation Program 10080619, funded by the Ministry of Trade, Industry and Energy (MOTIE, Korea) and CAD tools were supported by IDEC.

Acknowledgments: The authors would like to thank the anonymous reviewers for their constructive and helpful comments and suggestions.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Menouar, H.; Tuncer, A. UAV-enabled intelligent transportation systems for the smart city: applications and challenges. *IEEE Commun. Mag.* **2017**, *55*, 22–88. [CrossRef]
2. Scaramuzza, D.; Achtelik, M.C.; Doitsidis, L.; Friedrich, F.; Kosmatopoulos, E.; Martinelli, A.; Gurdan, D. Vision-controlled micro flying robots: from system design to autonomous navigation and mapping in GPS-denied environments. *IEEE Robot. Autom. Mag.* **2014**, *21*, 26–40.
3. Gatica, Z.N.; Munoz, P.C.; Sellado, A.P. Real fuzzy PID control of the UAV AR. Drone 2.0 for hovering under disturbances in known environments. In Proceedings of the 2017 CHILEAN Conference on Electrical, Electronics Engineering, Information and Communication Technologies (CHILECON), Pucon, Chile, 18–20 October 2017; pp. 1–6.
4. He, R.; Prentice, S.; Roy, N. Planning in information space for a quadrotor helicopter in a GPS-denied environment. In Proceedings of the 2008 IEEE International Conference on Robotics and Automation, Pasadena, CA, USA, 19–23 May 2008; pp. 1814–1820.
5. Arreola, L.; de Oca, A.M.; Flores, A.; Sanchez, J.; Flores, G. Improvement in the UAV position estimation with low-cost GPS, INS and vision-based system: Application to a quadrotor UAV. In Proceedings of the 2018 International Conference on Unmanned Aircraft Systems (ICUAS), Dallas, TX, USA, 12–15 June 2018; pp. 1248–1254.
6. Rawashdeh, N.A.; Rawashdeh, O.A.; Sababha, B.H. Vision-based sensing of UAV attitude and altitude from downward in-flight images. *J. Vib. Control* **2017**, *23*, 827–841. [CrossRef]

7. Fowers, S.G.; Lee, D.J.; Tippetts, B.J.; Lillywhite, K.D.; Dennis, A.W.; Archibald, J.K. Vision aided stabilization and the development of a quad-rotor micro UAV. In Proceedings of the 2007 International Symposium on Computational Intelligence in Robotics and Automation, Jacksonville, FL, USA, 20–22 June 2007; pp. 143–148.
8. Gageik, N.; Strohmeier, M.; Montenegro, S. An autonomous UAV with an optical flow sensor for positioning and navigation. *Int. J. Adv. Robot. Syst.* **2013**, *10*, 341. [CrossRef]
9. Ahrens, S.; Levine, D.; Andrews, G.; How, J.P. Vision-based guidance and control of a hovering vehicle in unknown, GPS-denied environments. In Proceedings of the 2009 IEEE International Conference on Robotics and Automation, Kobe, Japan, 12–17 May 2009; pp. 2643–2648.
10. Achtelik, M.; Bachrach, A.; He, R.; Prentice, S.; Roy, N. Autonomous navigation and exploration of a quadrotor helicopter in GPS-denied indoor environments. In Proceedings of the First Symposium on Indoor Flight, Mayagüez, Puerto Rico, July 2009.
11. Tomasi, C.; Kanade, T. Detection and Tracking of Point Features. *Int. J. Comput. Vis.* **1991**, *9*, 137–154.
12. Shi, J. Good features to track. In Proceedings of the 1994 IEEE Conference on Computer Vision and Pattern Recognition, Seattle, WA, USA, 21–23 June 1994; pp. 593–600.
13. Siong, L.Y.; Mokri, S.S.; Hussain, A.; Ibrahim, N.; Mustafa, M.M. Motion detection using Lucas Kanade algorithm and application enhancement. In Proceedings of the 2009 International Conference on Electrical Engineering and Informatics, Bangi, Malaysia, 5–7 August 2009; pp. 537–542.
14. Antonakos, E.; Alabort-i-Medina, J.; Tzimiropoulos, G.; Zafeiriou, S.P. Feature-based lucas-kanade and active appearance models. *IEEE Trans. Image Process.* **2015**, *24*, 2617–2632. [CrossRef] [PubMed]
15. Pearce, S.; Ljubičić, R.; Peña-Haro, S.; Perks, M.; Tauro, F.; Pizarro, A.; Dal Sasso, S.F.; Strelnikova, D.; Grimaldi, S.; Maddock, I.; et al. An Evaluation of Image Velocimetry Techniques under Low Flow Conditions and High Seeding Densities Using Unmanned Aerial Systems. *Remote. Sens.* **2020**, *12*, 232. [CrossRef]
16. Ma, S.; Bai, X.; Wang, Y.; Fang, R. Robust Stereo Visual-Inertial Odometry Using Nonlinear Optimization. *Sensors* **2019**, *19*, 3747.
17. Ci, W.; Huang, Y. A robust method for ego-motion estimation in urban environment using stereo camera. *Sensors* **2016**, *16*, 1704. [CrossRef] [PubMed]
18. Auysakul, J.; Xu, H.; Pooneeth, V. A hybrid motion estimation for video stabilization based on an IMU sensor. *Sensors* **2018**, *18*, 2708. [CrossRef] [PubMed]
19. Bouguet, J.Y. Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm. *Int. Corp.* **2001**, *5*, 4.
20. Available online: https://www.xilinx.com/support/documentation/ip_documentation/div_gen/v5_1/pg151-div-gen.pdf (accessed on 28 September 2020).
21. Cho, J.; Jung, Y.; Kim, D.S.; Lee, S.; Jung, Y. Moving object detection based on optical flow estimation and a Gaussian mixture model for advanced driver assistance systems. *Sensors* **2019**, *19*, 3217. [CrossRef] [PubMed]
22. Available online: https://www.xilinx.com/support/documentation/selection-guides/virtex7-product-table.pdf (accessed on 28 September 2020).
23. Ham, Y. C.; Shi, Y. Developing a smart camera for gesture recognition in HCI applications. In Proceedings of the 13th IEEE International Symposium on Consumer Electronics (ISCE), Kyoto, Japan, 25–28 May 2009; pp. 994–998.
24. Jang, W.; Oh, S.; Kim, G. A hardware implementation of pyramidal KLT feature tracker for driving assistance systems. In Proceedings of the 2009 12th International IEEE Conference on Intelligent Transportation Systems, St. Louis, MO, USA, 4–7 October 2009; pp. 1–6.
25. Chai, Z.; Shi, J. Improving KLT in embedded systems by processing oversampling video sequence in real-time. In Proceedings of the 2011 International Conference on Reconfigurable Computing and FPGAs, Cancun, Mexico, 30 November–2 December 2011; pp. 297–302.
26. Available online: https://www.dji.com/phantom-4 (accessed on 28 September 2020).
27. Frietsch, N.; Meister, O.; Schlaile, C.; Seibold, J.; Trommer, G. Vision based hovering and landing system for a vtol-mav with geo-localization capabilities. In Proceedings of the AIAA Guidance, Navigation and Control Conference and Exhibit, Honolulu, HI, USA, 18–21 August 2008; p. 6669.