2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP) | 978-1-6654-8308-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/ASAP54787.2022.00031

High-Performance AKAZE Implementation Including Parametrizable and Generic HLS Modules

Matthias Nickel, Lester Kalms, Tim Häring and Diana Göhringer
Technische Universität Dresden
Chair of Adaptive Dynamic Systems
Dresden, Germany
{Matthias.Nickel, Lester.Kalms, Tim.Haering1 and Diana.Goehringer}@tu-dresden.de

***Abstract*—The amount of image data to be processed has increased tremendously over the last decades. One major computer vision task is the extraction of information to find patterns in and between images. One well-studied pattern recognition algorithm is AKAZE which builds a nonlinear scale space to detect features. While being more efficient compared to its predecessor KAZE, the computational demands of AKAZE are still high. Since many real-world computer vision applications require fast computations, sometimes under hard power and time constraints, FPGAs became a focus as a suitable target platform. This work presents a highly modularized and parameterizable implementation of the AKAZE feature detection algorithm integrated into HiFlipVX, which is a High-Level Synthesis library based on the OpenVX standard. The fine granular modularization and the generic design of the implemented functions allows them to be easily reused, increasing the workflow for other computer vision algorithms. The high degree of parameterization and extension of the library enables also a fast and extensive exploration of the design space. The proposed design achieved a high repeatability and frame rate of up to 480 frames per second for an image resolution of 1920x1080 compared to related work.**

***Index Terms*—AKAZE, FPGA, HLS, Feature Detection, Computer Vision, Modular, Parametrizable**

I. Introduction

The use of sensors in industry, digital cameras and smartphones, which are more and more capable of capturing high-quality images and videos, increased. Therefore, the amount of processed image data increased tremendously over the last few decades and is still increasing rapidly. One major computer vision task performed on these kinds of data is the recognition of patterns. Feature detection and description are key parts of pattern recognition algorithms. One important measure to evaluate the quality of a feature detector is the repeatability. Repeatability in context of computer vision tasks is the capability of detecting the same features under different kind of transformations, such as change of viewpoint or scale. Two popular scale-invariant feature detector and descriptor are Scale Invariant Feature Transform (SIFT) [1] and Speeded Up Robust Features (SURF) [2]. While SURF has a lower computation time compared to SIFT by using the Determinant of Hessian (DoH) and the integral image instead of the Difference of Gaussian (DoG) for feature detection, both algorithm use floating point descriptor for features. Because floating point feature descriptor are in general computational expensive as well as have a high matching time for features, feature detection and description algorithms with binary descriptors like KAZE [3], Accelerated-KAZE (AKAZE) [4], Binary Robust Independent Elementary Features (BRIEF) [5], Oriented FAST and Rotated BRIEF (ORB) [6], Binary Robust Invariant Scalable Keypoints (BRISK) [7] and Fast Retina Keypoint (FREAK) [8] were proposed. This allows faster feature descriptor computation and much faster matching using typically the easy to compute Hamming Distance due the fact that other brute force matching metrics for binary descriptors show no significant differences [9]. With exception of KAZE, all these algorithms showed in general a much better improvement in the feature detection and description time compared to SIFT and SURF. The computational expensive nonlinear diffusion used by KAZE for feature detection performs in general worse than SURF, in some cases even worse than SIFT [10]. To improve the feature detection time using nonlinear diffusion the successor of the KAZE algorithm, named AKAZE uses the Fast Explicit Diffusion (FED) scheme for nonlinear diffusion instead of the Additive Operator Splitting (AOS) scheme. AKAZE also introduces downsampling of the image for nonlinear diffusion, building a nonlinear scale space pyramid. The AKAZE algorithm showed better results in context of repeatability and accuracy in comparison to SIFT, SURF, ORB and BRISK. In some cases, AKAZE showed even better repeatability and accuracy than its predecessor KAZE [3], while at the same time being faster than SURF. However, it is still more computational expensive and therefore slower in comparison to ORB and BRISK.

In order to make the AKAZE algorithm feasible to use for real-time applications as well as in data center or in the field of High-Performance-Computing (HPC), this paper presents a hardware acceleration of the feature detection part. This proposed hardware friendly feature detector is implemented in a streaming fashion using line-buffers making it less dependent on the image size in the context of scalability. The feature detector of this implementation is complemented with the FREAK feature descriptor, which has shown to result in a faster computation with lower memory load compared to the original Modified-Local Difference Binary (MLDB) descriptor. This work is also based on a highly modular High-Level Synthesis (HLS) based computer vision library showing a use case how easily a performant implementation of a complex computer vision algorithm can be implemented on a Field

2160-052X/22/$31.00 ©2022 IEEE
DOI 10.1109/ASAP54787.2022.00031

Programmable Gate Array (FPGA).

This work is structured as follows, Section II presents information about related work. Section III gives a brief overview of the AKAZE algorithm while Section IV presents implementation specific details and differences compared to the original algorithm. Section V shows the achieved results of the implementation compared to related work and Section VI summarizes this paper and gives a brief outlook.

II. Related Work

Over the years since the publication of the original AKAZE algorithm by Alcantarilla et al. in 2013 [4], multiple works on various architectures with the focus on different optimization methods have been proposed.

In 2017, Kalms et al. [11] presented an implementation of the AKAZE feature detector, including the build of the nonlinear scale space pyramid with two octaves on an FPGA complemented with a FREAK descriptor implementation in software. The hardware implementation uses a 16-bit fix point datatype except for the computation of the DoH which uses a 32-bit fix point datatype. The design was evaluated on a Zedboard containing a Zynq-7000 SoC with 100 MHz, achieving 98 frames per second (fps) for an image resolution of 1024×768. In 2019, Kalms et al. [12] extended the design by an FPGA implementation of the FREAK descriptor achieving 73.4 fps for 2048 features, showing that much higher framerates can be achieved with an improved DMA system for random memory access.

Jiang et al. [13], [14] introduced a real-time hardware friendly oriented implementation of the AKAZE feature detector, and a descriptor called Polar Local Difference Binary (PLDB). The implementation verified on a Virtex-5 and mapped to a 65 nm TSMC ASIC achieved 127 fps for an image size of 1920×1080 pixel at a frequency of 200 MHz. However, their implementation does not include the calculation of the contrast factor which is an important part of the AKAZE algorithm.

In 2018, Mentzer et al. [15] proposed an implementation of the AKAZE algorithm for the Cadence Tensilica Vision P5 an Application-Specific Instruction-set Processor (ASIP). Because the implementation was targeted for the stereo camera calibration for an Advanced Driver Assistant System (ADAS) and the produced stereo images do not much differ in scale or orientation, several simplifications of the AKAZE algorithm were done, leading to a great reduction of the feature vector size. The implementation reached up to 20.3 fps for an unoriented and 15.5 fps for an oriented descriptor for 800×640 images at a maximum clock frequency of 800 MHz.

Li et al. [16] proposed an implantation of the AKAZE algorithm using the SIFT descriptor, combining the efficiency of the AKAZE feature detector with stability and robustness of the SIFT descriptor. To reduce high dimensionality and therefore inherent high computational complexity, the SIFT descriptor of this implementation is based on a Sparse Random Projection (SRP). While improving the quality of the descriptor compared to MLDB and having much faster matching times compared to the standard SIFT descriptor, SRP-AKAZE has also higher matching time compared to MLDB, making it a compromise between MLDB and SIFT descriptor.

Du et al. [17] uses Hardware Friendly Descreening (HFD) [18] as diffusion filter for the linear scale space. To further increase the processed framerate, the first octave of the following frame is predicted using the current frame. To counter the effect of the reduction of robustness caused by this prediction they use motion estimation to predict the motion between the previous and current frame. However, even with this method the accuracy compared to the original AKAZE is reduced. Still, their implementation can be beneficial for applications which require a high number of frames to be processed were the changes between frames is quite small. They achieved 784 fps for 640×480 image for their design containing the build of nonlinear scale space, feature detection, description and matching.

Soleimani et al. [19] proposed an AKAZE implementation for FPGAs using image partitioning to compute different sections of the image in parallel. They also store the conduction coefficient matrix, which is required by the FED cycle, as well as the diffused image on-chip in the partitioned Block-RAM (BRAM) for the parallel computation. The conduction coefficients for the following step are directly calculated by the output of the FED cycle of the previous step. This simultaneous pipelined design allows to overlap the computation of the FED function and conduction coefficient matrix. They achieve with this design 304 fps for a 1280×720 8-bit grey scale image. The downside is that the storing of the conduction coefficient matrix and the defused image is equivalent of storing the whole image two times on board. As a result, this design is highly reliant on the available on-chip memory, making it heavily dependent on the image size and therefore limiting its scalability. Their proposed design only contains the computation of the contrast factor and the construction of the nonlinear scale space, but neither the feature detection nor the description.

III. AKAZE

AKAZE is a fast multiscale feature detection and description algorithm based on the KAZE algorithm. Both AKAZE and KAZE build a nonlinear scale space for feature detection. AKAZE uses a scheme called FED, unlike the original KAZE, which uses the AOS scheme instead. Both schemes are semi-implicit schemes to compute the nonlinear diffusion. Unlike AOS, FED allows a variable time step size that can violate the stability condition during an FED cycle. However, it guarantees that a stable state is reached at the end of each cycle. The second difference of AKAZE compared to its predecessor KAZE is that it uses the MLDB descriptor instead of the Modified-SURF (M-SURF) descriptor.

The AKAZE feature detector can be divided into three parts: the computation of the contrast factor, the building of the nonlinear scale space pyramid, and the detector that computes the DoH for the feature response values.

The computation of the contrast factor is based on the noise estimator proposed by Canny [20]. A global histogram of noise amplitudes, which are the absolute gradient values of the image, is build and a fixed percentile of the noise signal is then used to estimate the noise strength. In AKAZE a 70% percentile of histogram consisting of 300 bins is used to determine the contrast factor.

The construction of the nonlinear scale space pyramid is discretized into a set of levels called evolution. These evolution levels are further organized in a series of octaves with each octave o consisting of exactly s sublevels. Each evolution level produces a smoothed image of the diffused image of the previous evolution level, using a Gaussian filter kernel. This Gaussian filtered image is used by the feature detector of each evolution level, as well as for the computation of the conduction coefficient matrix, required by the FED. The first FED step of the FED cycle takes the diffused image of the FED cycle of the previous evolution level and produces a further diffused image for the following evolution level. Each FED cycle consists of multiple steps with variable step sizes. The step sizes for all FED steps are independent of the input image, since they depend only on the number of octaves and sublevels of the nonlinear scale space pyramid, and can therefore be calculated in advance. Since images can differ in their luminance values and may also contain noises, a contrast factor is required to calculate the conduction coefficients. There are multiple options with different properties to compute the conduction coefficients for the nonlinear diffusion. One popular option is the anisotropic diffusion, also called Perona-Malik diffusion [21], which is not the only one supported by AKAZE. To make the feature detector scale invariant, the image is downsampled by the factor of two at the end of each octave. This produces an image with only a quarter of the size of the previous octave. In addition, an adaption of the contrast factor by multiplying it with 0.75 is required.

IV. Proposed Design

Two goals are pursued with this work. On the one hand, a modular and scalable implementation of the AKAZE algorithm that can be optimized in terms of performance, resources and repeatability, depending on the target device and application. On the other hand, generic and parameterizable building blocks were realized with HLS to improve the implementation of different feature detection algorithms. Therefore, the individual function parameters and options have been designed in such a way that they can cover a wide range of potential applications. For some functions needed in the algorithm we used and extended an already existing HLS-based library [22].

A. Fixed Point

The original implementation of the AKAZE algorithm used 32-bit floating-point precision. Because floating point data types are expensive to implement in hardware, the design supports 8- and 16-bit fixed-point data types for the images.

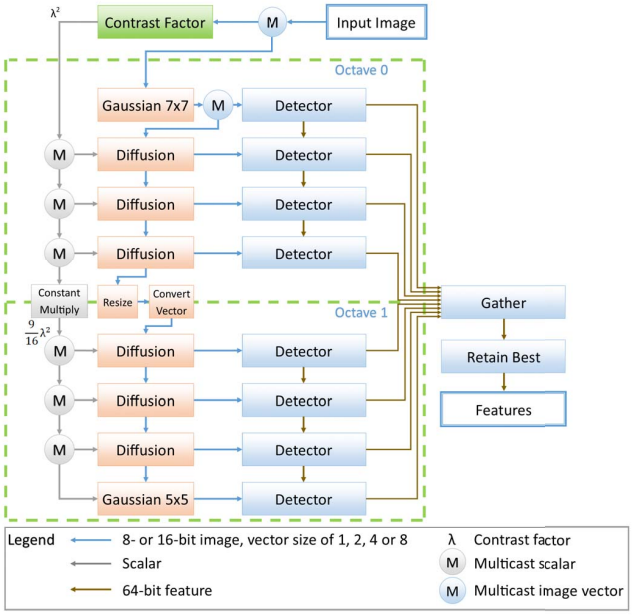


Fig. 1: Overview of the proposed AKAZE feature detector design, including the build of the nonlinear scale space pyramid and contrast factor computation.

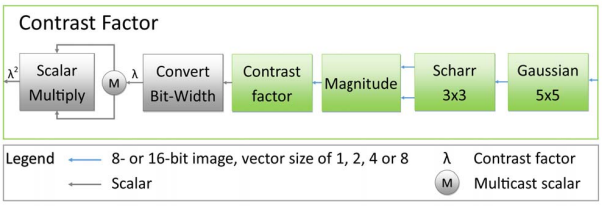


Fig. 2: Modularization of the contrast factor computation using Scharr and Magnitude function to compute the absolute gradient.

The only exception is the DoH function, which creates 16- or 32-bit response values used by the feature extraction function.

B. Vectorization

All functions of the implementation provide a parallelization or vectorization of 1, 2, 4 or 8, to achieve the desired performance based on the available resources. This includes the computation of the contrast factor, the nonlinear scale space, and the feature detection. Only the Gather and Retain Best functions (see Figure 1), do not require parallelization. This is partly due to their complexity and partly because they operate only on feature vectors, which are much smaller than the images.

C. Contrast Factor

The contrast factor computation can be divided into four major parts, as shown in Figure 2. A 5×5 Gaussian kernel is used to smooth the input image. The first order derivatives are computed in parallel with respect to x and y and than used by the magnitude function to compute the absolute gradients.

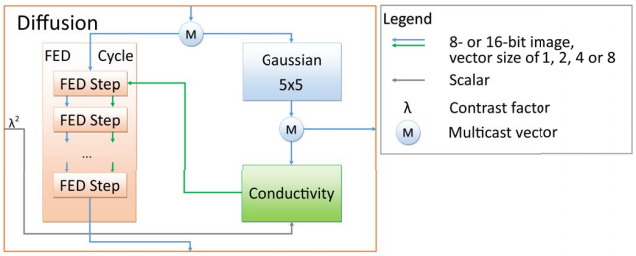


Fig. 3: Overview of the nonlinear diffusion. The input image is multicasted to the FED Cycle and a Gaussian filter. The Gaussian image is then multicasted to the conductivity function and to the feature detector.

Finally, the absolute gradient values are evaluated by the contrast factor function itself which includes the creation of a histogram.

Figure 2 describes the complete implementation of the contrast factor computation. In this context, we computed λ as shown in Equation 1. The main part of the contrast factor implementation is the calculation of the histogram, which was realized by a double buffer. Like in Kalms et al. [11] the calculation is done in one loop instead of two. This is possible because the maximum achievable value of the gradient image ($h_{max} = 0.5$) is used for the histogram size and not the maximum achieved value of each gradient image. Furthermore, the size of the histogram (b_{max}) was set to 256 for 8-bit and 512 for 16-bit input data to achieve a sufficient accuracy and optimized utilization of the local memory.

$$\lambda = h_{max} \cdot \frac{i}{b_{max}} = \frac{i}{2b_{max}} \tag{1}$$

As in the original algorithm, this design uses a 70% percentile for the contrast factor, but a different percentile can be freely chosen if required. For a vectorization of n, n histograms are computed in parallel. These histograms are then summed together with the calculation of the bin position (i) until the percentile has been reached. Since we only need the squared contrast factor in the rest of the algorithm, we calculate it in advance to avoid unnecessary operations. This additionally requires a bit-width conversion from 16 to 32-bit and a multicast in hardware.

D. Nonlinear Scale Space Pyramid

The computation of the contrast factor provides the basis for the nonlinear diffusion. As shown in Figure 3, the nonlinear diffusion can also be modularized into three parts. First, the 5×5 Gaussian filter which takes the image produced by the previous evolution level as input and writes the output to the feature detector, which will be discussed later. Second, the conductivity function, which produces the conduction coefficients required by the FED steps. Third, an FED cycle consisting of multiple FED steps. Since the Gaussian filter function already existed, we implemented the conductivity and FED function. Both functions were not implemented from scratch. Instead they were used as an opportunity to extend the already existing 2D filter function which is responsible for handling the other filters like the Gaussian or Scharr filter. This allowed to skip time expensive (re-) implementation of often used data handling functionality, like line-buffers, filter windows and vectorization which already existed in the library and are highly optimized for hardware synthesis.

The conductivity function uses the more hardware friendly optimization presented in [11]. Requiring just one instead of two divisions as described in the following equation:

$$c(\nabla L(x,y)) = \frac{1}{1 + \left(\frac{|\nabla L(x,y)|}{\lambda}\right)^2} = \frac{\lambda^2}{\lambda^2 + |\nabla L(x,y)|^2} \tag{2}$$

The implementation reused the already optimized derivative functions with respect to x and y which exploits the symmetry of filter kernels like Scharr and Sobel to compute the square gradient value. However, the synthesis tool had problems to identify the required number of bits for computing the square of the first order derivatives with respect to x and y, allocating resources for a bit-width much higher than required. Therefore, a vendor independent arbitrary precision data type, supporting a bit-width of up to 64 bit to address this issue, was realized.

Tests done in software using the Affine Covariant Regions Dataset of the University of Oxford [23] showed that only 8 of over 400 million computed pixels by the FED function exceeded a value larger than one or lower than zero. This led to the decision to saturate the exceeding values to the maximum representable value (using 8- or 16-bit) or to set them to zero if they would become negative. This increases the fraction of the fixed-point representation by two bits to increase the accuracy. The FED function requires two inputs, the image and the conduction coefficients, where the latter is the same for all FED steps in a cycle. Therefore, the conduction coefficients are simultaneously forwarded together with the FED output by a newly introduced forwarding function, saving memory resources for buffering. To support these new requirements, the generic filter function has been adapted to optionally have up to two inputs and two outputs.

The resize function is used to reduce the image columns and rows by a factor of 2 by averaging four neighboring pixels, which is a fast area interpolation. Factors of 4 and 8 are also supported. Because the design averages always 2^n with $n \in \{2, 4, 8\}$ pixels a hardware friendly implementation is possible using only addition and shifting, instead of a costly division. The implementation of the resize function was designed in a generic way not only to support averaging of pixels but also to support further interpolation methods. Based on the 2D filter function the resize function also supports a vectorization of the input of up to 8. Because for every 2^n input pixels only one output pixel is created, the output vector of v_{out} is the quotient of the input vector v_{in} and the downsampling factor n. Since for a downsampling factor of 2 an output is written only for every second input row, an additional data width converter is used to halve the vector size if possible. This aligns the vector size with the image size to avoid unnecessary stalls.

Analogous to the original AKAZE feature detection algorithm, the contrast factor is also adapted by a factor of 0.75.

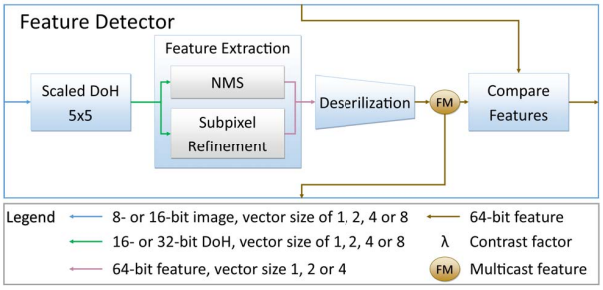


Fig. 4: Overview of the feature detector module. The detected features are multicasted to the compare function of the current and following sublevel. NMS = Non-Maxima Suppression

As this implementation uses the square of the contrast factor, it is adapted by a constant multiplication with a factor of $0.5625 = 1/2 + 1/16$, which can be realized by two shifts and one addition instead of a more expensive multiplication.

E. Feature Detector

The feature detector consists of three parts in the original AKAZE algorithm. The DoH response values are computed using 3×3 Scharr filter kernels with the corresponding scale. The scaled kernels are applied on the diffused image to compute the first order derivatives with respect to x and y. In addition, the same scaled kernels are applied on the first order derivatives again to compute the second order derivatives with respect to xx, yy and xy. The second order derivatives are then used to compute the DoH response value. The second part is the detection of the scale space extrema by extracting the largest response value in a 3×3 window and compare it to all other detected extrema in its radius to find the largest extrema or in other words, the most significant feature in a region. Since pixel location is not sufficient for many cases and features can be found between pixels, a subpixel refinement is applied on the remaining features as last part. Because a straight forward approach using the same concept would not be efficient for a direct implementation in hardware a restructuring of this design is proposed in Figure 4.

Software implementations can access the same memory location multiple times as well as adjacent memory. However, this approach is not feasible for a streaming based hardware accelerator implementation. Instead an upscaled kernel window is required which covers the whole required area. Therefore, a new compile time solvable scale function which automatically scales up the kernel by filling the gaps with zeros for the kernel is introduced. The compiler is able to recognize all multiplications with zero and optimizes them out, so that no overhead in the resource allocation for DSPs is created. The only disadvantage is that with the increasing kernel size, in addition to the kernel windows size also the number of line-buffer lines increases, putting higher demands on the on-chip-memory resources. To reduce the resource demands as well as the latency the 3×3 Scharr filter kernel for the first order derivatives with respect to x and y were convolved into three 5×5 filter kernel to compute the second order derivatives directly. This led to the introduction of a compile time solvable kernel convolution function, which can convolve any two given input kernels with each other. As the three created kernels also have symmetrical properties which can be exploited, three second order derivative functions are introduced similar to the already existing first order derivative functions. Compared to a separate implementation using two Scharr filter modules $1/3$ of BRAM utilization could be saved.

A feature extraction function that compares all detected features of the same level with those of the previous level would be possible but would have a large disadvantage in hardware. Since the design reads in a streaming fashion, as soon as one feature is detected and compared to the others, it would stop reading and stall the entire design during the comparison phase. For this reason, and because it also matches with the fine granular idea of the used library, the scale space detection function used in the original AKAZE implementation is divided in two modules, named feature extraction and feature comparison. The subpixel-refinement done in the post-detection part also uses the DoH response values to find the subpixel location. This led to the decision to modularize the feature extraction into a Non-Maxima Suppression (NMS) module and a subpixel refinement module.

Another difference compared to the original AKAZE algorithm is that the NMS function, which is an existing library function, follows the NMS definition of the OpenVX standard [24]. According to the definition the top-left elements have to be smaller or equal and the bottom-right elements only smaller compared to the center element. In the original algorithm all eight surrounding DoH response values have to be smaller.

Because NMS and subpixel-refinement are using the same input, both are computed in parallel regardless of whether the NMS function has rejected the pixel or not. Only if both modules accept the response value as a valid feature and if it is beyond a certain threshold, it will be written to the output. Otherwise it will be discarded.

To keep the library generic, not only is the subpixel refinement, which is quite computational and resource demanding, optional, but also the NMS. In addition, the functions for creating features and for subpixel refinement are exchangeable. By using function pointers to realize the easy exchangeability, the implementation exploits high-level language features provided by C/C++ and uses a wrapper for AKAZE feature detector specific functions. This allows to adapt the detector depending on the algorithm and characteristic of input data as well as of the created feature.

Since more than one feature can be detected for one input vector the feature extraction function can also write feature vectors to the output, containing more than one element. In case that not for every entry in the output vector a feature has been detected, an invalid element is written to the according position in the vector. However, it is ensured that each output vector contains at least one valid feature. The deserialization function disassembles the output vector of the feature extraction function and writes out all valid vector entries one by one and discards all invalid entries. The deserialization function

supports replacing its functionality to decide which entry is valid or not.

Unlike the multicast function for images, the number of features is unknown at compile/synthesis time, thus requiring a multicast function that can handle an arbitrary number of features. Therefore, a new feature multicast function has been introduced, stopping multicasting if an invalid input vector has been detected, which signals the end of a transaction by a feature function.

Like in the original AKAZE algorithm all neighboring features are compared against each other to avoid replication. The design of the comparison function allows to read between one and three inputs and compares the features of all these inputs against each other. The original AKAZE implementation compares not only the features of the current sublevel against each other but compares them also against the features of the previous and following sublevel of an octave. Therefore, this design also presents a comparison function with the support of up to three inputs. However, as Kalms et al. [11] evaluated for AKAZE, the lower sublevel contributes to only about 1% of the matches. Therefore, we have also omitted this in our final design. The features of the feature extraction function of the current and previous sublevel are read from the input and compared against the features already stored in the buffer and then written to the end of the buffer. Due to the fact that the features are detected in ascending order with respect to the vertical location of the image, the comparison is stopped as soon as the first feature stored in the buffer is outside of the vertical radius. Because the detectable features inside the radius are in general much lower than the overall features detected by the feature extraction function, the design automatically limits the buffer size to the worst-case number of detectable features in the radius. The buffer is therefore designed in a ring-buffer fashion. This means that as soon as the maximum capacity of the buffer is reached, the oldest stored feature in the buffer is replaced by the newest one. Depending on whether the replaced feature is still valid or not, it is written to the output or discarded. The same mentality driven to keep the feature extractor generic also led to a generic implementation of the compare function. All functionalities concerned with the specific characteristics of the features are also exchangeable and wrapper functions can be used to specialize the function, as has been done for our AKAZE implementation.

F. Additional Functions

We developed a gather function that supports both block or cyclic mode. This was designed in conjunction with a scatter function. Both functions require that the number of elements to be scattered or gathered is known at compile time. Due to the fact that the number of detectable features is unknown, a feature gather function was implemented. Similar to the feature multicast function this gather function also stops reading from an input as soon as an invalid feature or the maximum defined number of elements for this input has been read. The gather function of the AKAZE feature detector is used to collect the features from all evolution levels, creating one output stream to be finally processed by the Retain Best function. The feature gather function also supports the exchange of the function that decides which element signals the end of a transaction. This allows various complex data types to be handled by this function.

V. Evaluation

This section provides the results and metrics of our implementation of the AKAZE feature detector algorithm. The evaluation of the repeatability for different configurations of our design was done in software using the Oxford affine covariant features dataset [23]. The image dataset contains 8 images with 5 additional transformations for each image, covering image transformations like zooming, rotation, change of viewpoint, blurring, change in brightness and JPEG compression. The software evaluation only uses the same modules as proposed for the hardware design. The final design was evaluated on the ZCU104 evaluation board featuring a Xilinx Zynq UltraScale+ MPSoC. We synthesized the modules using a graph-based tool based on the OpenVX standard, complemented with Vivado HLS 2020.1 for the IP-cores and Vivado 2021.1 for the synthesis of the block-design. The design consist of 95 IP-cores for two octaves and 163 IP-cores for three octaves.

A. Comparison with other FPGA implementations

Table I shows the overall resource utilization, achieved frequency and frame rate for our design compared to related work. To make it more comparable, we also present the image resolution, the stages implemented in hardware, and whether the design was tested in hardware or in simulation. The frame rate of our design is with 480 fps higher than the others. The only exception is Du et al. [17] with 784 fps. However, the image size of Du et al. [17] is only 640×480, while ours is 1920×1080 which are 6.75 times the number of pixels to compute. With the same resolution we achieve 1998 fps. Soleimani et al. [19] achieved a frame rate of 304 fps, but at a resolution of 1280×720. Because their work only focused on the nonlinear scale space they are missing the feature detector. Their implementation uses buffers two times the image size to store the conduction coefficients and diffused FED image on-chip. Therefore, their Memory Management Unit (MMU) already requires 524 BRAM, which strongly restricts the scalability of their design. While Jiang et al. [13] achieved a similar frequency to our design at the same frame resolution, at only 127 fps, the frame rate is much lower than ours. Their implementation also does not include the computation of the contrast factor of an image. Kalms and Göhringer [25] presented an OpenCL based FPGA (Virtex-7 XC7VX690T) implementation of the AKAZE feature detector. The Scharr, Gaussian, conductivity, FED, DoH and the feature extraction kernel function did run on the FPGA while the comparison functions for the extracted features did run on the host CPU. They achieved 342 fps (709 Million Pixels per Second (Mpps)) with a frequency of 200 MHz and an images size of 1920×1080 with their design. Which is close to our

	Our Design		Kalms et al.[11]	Soleimani et al.[19]	Du et al.[17]	Jiang et al.[13]
Device	Zynq®UltraScale+™ (ZCU104)		Zynq®-7000 SoC (7020)	Kintex®UltraScale™ (KCU105)	Kintex®-7 (XC7K325T)	ASIC (TSMC 65nm)
Image resolution	1920x1080		1024x768	1280x720	640x480	1920x1080
Contrast factor	✓		✓	✓	?	✗
Nonlinear scale space	✓		✓	✓	✓	✓
Detector	✓		✓	✗	✓	✓
Descriptor	✗		✗	✗	✓	✓
Matching	✗		✗	✗	✓	✗
Test on device	✓		✓	✗	✓	✓
FF	79 835	123 944	38 314	65 028	157 122	-
LUT	69 722	127 063	24 945	112 596	196 134	-
LUTRAM	-	-	-	72 276	28 068	-
36k BRAM	147.5	232	108	524	291	-
URAM	4	4	-	-	-	-
DSP	136	272	157	31	228	-
Frequency (MHz)	214.5	150	100	100	100	200
Frames per second	360	480	98	304	784	127
Million pixels per second	746	996	77	280	241	263

TABLE I: Proposed design with a vector size of 4 (left) and 8 (right) compared to related work. All designs presented in this table implemented 2 octaves.

proposed design. However, it is to consider that the design in [25] consists of only one octave.

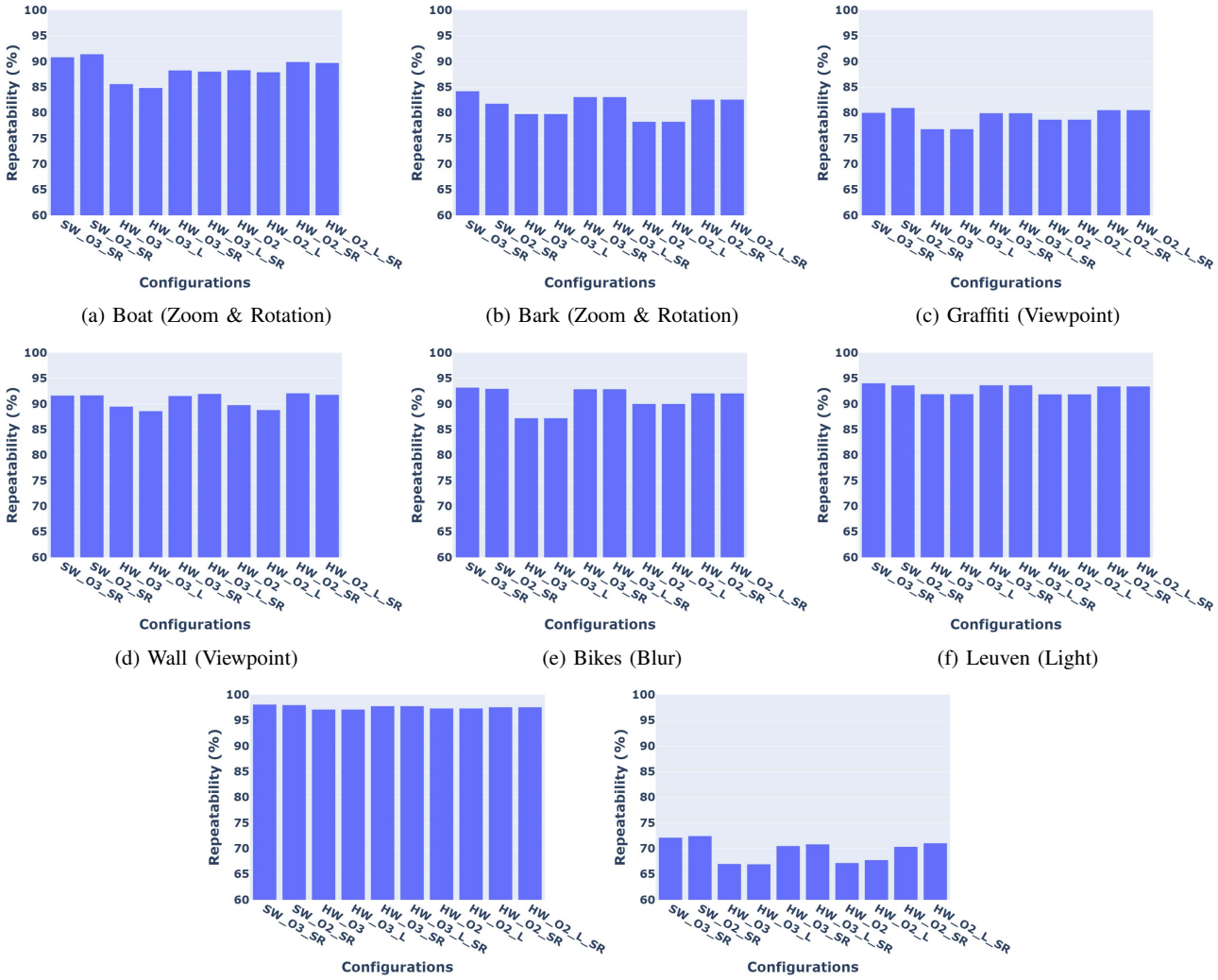
B. Comparison with CPU and GPU implementations

Pieropan et al. [26], [27] implemented AKAZE (as well as SIFT) in CUDA. They used a Intel Xeon processor with 16 cores at 2.6 GHz, with 32 GB of RAM and an Nvidia Titan X (Maxwell) for their evaluation, doing 100 runs on an image of the Iguazu data set [4] with a resolution of 900×675 pixels. For the feature detector they achieved an average frame rate of around 154 fps (94 Mpps) on the GPU and 8.5 fps (5.2 Mpps) on the CPU. Since the building of nonlinear scalar space of the AKAZE algorithm requires many sequential steps, they concluded that it is not really suitable for a GPU architecture. In addition to the OpenCL based FPGA implementation, Kalms and Göhringer also [25] presented a OpenCL implementations of the AKAZE feature detector for a CPU (Intel Core-i7 4770k), integrated GPU (iGPU) (Intel HD 4600) and GPU (Nvidia GTX 780). They achieved with an images size of 1920×1080 pixels, 27 fps (56 Mpps) on the CPU, 32 fps (67 Mpps) on the iGPU and 233 fps (483 Mpps) on the GPU. Our proposed design is around 2.3 times faster than the GPU implementation of [26] and around 1.5 faster than the GPU implementation of [25]. However, it is to consider that [26] implemented 4 octaves and [25] implemented 1 octave. While performance comparisons between different architectures are hard to make, we can show that thanks to the streaming based nature of our design and the exploitation of pipeline and data parallelism (vectorization) an FPGA implementation of the AKAZE feature detector can outperform a GPU implementation.

C. Repeatability

We tested our design in both hardware and software, using 8-bit gray scale images and 16-bit for DoH computation. However, image processing using 16-bit fixed-point precision and 32-bit for DoH response values is also supported. We complemented our hardware implementation of the AKAZE feature detector with a software implementation of the FREAK [8] descriptor for evaluation. To show the correctness of our design, we decided to use the repeatability [23] as metric for evaluation. The repeatability is the ratio of detected features that survive photometric and geometric transformations. It is calculated on the basis of the intersection of regions and depends only on the detector and not on the descriptor. Therefore, making it a good choice for our evaluation of our detector implementation.

One major contribution of our work is the implementation of a fully pipelined compare function. Using a ring buffer design to store the intermediate results for comparison and the fact that features are sorted with respect to the y position, reduced the complexity from $\mathcal{O}(n^2)$ to approximately $\mathcal{O}(n \log b)$, with b denoting the ring buffer size and n the number of input features to compare. The ring-buffer size is computed automatically depending on the number of detectable features in the search radius, but it can also be manually restricted if required. However, the number of maximum required loop iterations can still become quite high. To integrate the compare function into the pipeline, its loop iterations should be limited. While rewinding the loop would be possible to ease the constraint, it would lead to a strong reduction of the frequency and increase the initiation interval. Therefore, we allow to define the maximum number of iterations manually. We tested our design for two and three octaves, with and without iteration limitation for the feature compare function, and with and without subpixel refinement in hardware. We compared it against a pure 32-bit floating point implementation done in software with subpixel refinement. The result for the repeatability are shown in the Figure 5. The 32-bit floating point implementation performs best. However, the hardware using subpixel refinement achieves close results to the software implementation, with differences between 0.05% and 2%. The only exception is the boat set (Figure 5a) with around 2.6%



(a) Boat (Zoom & Rotation) (b) Bark (Zoom & Rotation) (c) Graffiti (Viewpoint)

(d) Wall (Viewpoint) (e) Bikes (Blur) (f) Leuven (Light)

(g) UBC (JPEG compression) (h) Trees (Blur)

Fig. 5: Repeatability a software implementation and different configurations for our proposed hardware design. SW = 32-bit floating-point, HW = 8-bit fixed point for images and 16-bit fixed-point for DoH, O2/3 = 2 or 3 octaves, L = Latency Restriction for compare function, SR = Subpixel Refinement using 32-bit floating-point for both SW and HW

for three octaves. However, they perform in all cases better compared to the hardware implementation without subpixel refinement, with up to 5.6% improvement in the bikes set (Figure 5e) for three octaves. The hardware has an average loss of less than 1% compared to software when subpixel refinement is selected. If we omit subpixel refinement in hardware, we lose on average additional 2.6% for three octaves and 1.6% for two octaves. This shows that a fixed-point hardware implementation has the potential to achieve similar results in context of repeatability compared to a 32-bit floating-point software implementation. The results also show that the difference in repeatability is negligible if the maximum number of iterations for the comparison module is carefully chosen. Therefore, we have chosen a value similar to that of the slowest function (DoH) of the respective level to not slow down the pipeline.

Stage	LUT	FF	BRAM	URAM	DSP
Constrast Factor	3772	3295	7	0	4
Nonlinear Scale Space	24351	27265	49	0	0
Detector	41599	49275	91.5	4	132
Inter Node FIFOs	7746	13524	0	39	0
System	7223	11634	7.5	0	0

TABLE II: Resource consumption of each stage of our hardware design with 2 octaves, vectorization of 4, no subpixel refinement and latency restrictions.

Table II shows a more detailed breakdown of the resource utilization. The system resources are needed for the connection to the main memory and the ARM processor. Since there was

plenty of unused URAM on the system, they were partly used for our inter node FIFOs, but normal BRAM would have been sufficient as well.

VI. Conclusion and Future work

In this work, we presented a hardware friendly design for the AKAZE feature detection algorithm using HLS. Our design is streaming based and fully pipelined. This allows an overlapping parallel computation of different modules. An utilization of line buffers, kernel windows, and FIFOs between the modules to store intermediate results, keeps the on-chip memory utilization low compared to other related work. The use of the template based HLS library allowed an easy implementation of many functions required by the algorithm. However, not all required functions could be realized at first. Therefore, this work extended the library with new functionalities while keeping them at the same time as generic as possible. The introduction of exchangeable functions using function pointers allows to extend the library much easier. Therefore, they can be reused for different algorithms not just for AKAZE. We showed that a hardware friendly implementation using HLS is possible, achieving 480 fps for a 1920×1080 resolution on a Zynq UltraScale+. The use of the HLS library reduced the high workload which would be required for a fine tuned implementation using a Hardware Description Language (HDL) like VHDL or Verilog. Designing a hardware-friendly subpixel refinement without the costly use of 32-bit floating point data is a prime target for future work, in addition to a hardware-friendly implementation of a descriptor like FREAK using HLS, and making the implemented functions open source.

Acknowledgment

The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany in the programme of "Souverän. Digital. Vernetzt.". Joint project 6G-life, project identification number: 16KISK001K

References

[1] D. G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *International Journal of Computer Vision (IJCV)*, pp. 91–110, 2004.

[2] H. Bay, T. Tuytelaars, and L. Van Gool, "SURF: Speeded Up Robust Features," in *European Conference on Computer Vision (ECCV)*, 2006, pp. 404–417.

[3] P. F. Alcantarilla, A. Bartoli, and A. J. Davison, "KAZE Features," in *European Conference on Computer Vision (ECCV)*, 2012, pp. 1–14.

[4] J. N. P. F. Alcantarilla and A. Bartoli, "Fast Explicit Diffusion for Accelerated Features in Nonlinear Scale Spaces," in *British Machine Vision Conference (BMVC)*, Sep. 2013, pp. 1–11.

[5] M. Calonder, V. Lepetit, M. Ozuysal, T. Trzcinski, C. Strecha, and P. Fua, "Brief: Computing a Local Binary Descriptor Very Fast," *Trans. Pattern Anal. Mach. Intell. (TPAMI)*, pp. 1281–1298, 2012.

[6] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: An Efficient Alternative to SIFT or SURF," in *International Conference on Computer Vision (ICCV)*, 2011, pp. 2564–2571.

[7] S. Leutenegger, M. Chli, and R. Y. Siegwart, "BRISK: Binary Robust Invariant Scalable Keypoints," in *International Conference on Computer Vision (ICCV)*, 2011, pp. 2548–2555.

[8] A. Alahi, R. Ortiz, and P. Vandergheynst, "FREAK: Fast Retina Keypoint," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012, pp. 510–517.

[9] E. Bostanci, "Is Hamming distance only way for matching binary image feature descriptors?" *Electronics Letters*, pp. 806–808, 2014.

[10] S. A. K. Tareen and Z. Saleem, "A comparative analysis of SIFT, SURF, KAZE, AKAZE, ORB, and BRISK," in *International Conference on Computing, Mathematics and Engineering Technologies (iCoMET)*, 2018, pp. 1–10.

[11] L. Kalms, K. Mohamed, and D. Göhringer, "Accelerated Embedded AKAZE Feature Detection Algorithm on FPGA (Best Paper)," in *International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART)*, 2017, pp. 1–6.

[12] L. Kalms, M. Hajduk, and D. Göhringer, "Efficient Pattern Recognition Algorithm Including a Fast Retina Keypoint FPGA Implementation," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 121–128.

[13] G. Jiang, L. Liu, W. Zhu, S. Yin, and S. Wei, "A 127 fps in Full HD Accelerator Based on Optimized AKAZE with Efficiency and Effectiveness for Image Feature Extraction," in *Design Automation Conference (DAC)*, 2015, pp. 1–6.

[14] G. Jiang, L. Liu, W. Zhu, S. Yin, and S. Wei, "A 181 gops akaze accelerator employing discrete-time cellular neural networks for real-time feature extraction," *Sensors*, pp. 22 509–22 529, 2015.

[15] N. Mentzer, J. Mahr, G. Payá-Vayá, and H. Blume, "Online Stereo Camera Calibration for Automotive Vision based on HW-accelerated A-KAZE-Feature Extraction," *Journal of Systems Architecture (JSA))*, pp. 335–348, 2018.

[16] D. Li, Q. Xu, W. Yu, and B. Wang, "SRP-AKAZE: An improved accelerated KAZE algorithm based on the sparse random projection," *IET Computer Vision*, pp. 131–137, 2020.

[17] S. Du, Y. Li, and T. Ikenaga, "Temporally Forward Nonlinear Scale Space for High Frame Rate and Ultra-Low Delay A-KAZE Matching System," *Transactions on Information and Systems IEICE*, pp. 1226–1235, 2020.

[18] H. Siddiqui, M. Boutin, and C. A. Bouman, "Hardware-Friendly Descreening," *Trans. Image Process.*, pp. 746–757, 2010.

[19] P. Soleimani, D. W. Capson, and K. F. Li, "Real-time FPGA-based implementation of the AKAZE algorithm with nonlinear scale space generation using image partitioning," *Journal of Real-Time Image Processing (JRTIP)*, pp. 2123–2134, 2021.

[20] J. Canny, "A Computational Approach to Edge Detection," *Trans. Pattern Anal. Mach. Intell. (TPAMI)*, pp. 679–698, 1986.

[21] P. Perona and J. Malik, "Scale-Space and Edge Detection Using Anisotropic Diffusion," *Trans. Pattern Anal. Mach. Intell. (TPAMI)*, pp. 629–639, 1990.

[22] L. Kalms and D. Göhringer, "Accelerated High-level Synthesis Feature Detection for FPGAs using HiFlipVX," in *Towards Ubiquitous Low-power Image Processing Platforms*, 2021, pp. 115–135.

[23] K. Mikolajczyk, T. Tuytelaars, C. Schmid, *et al.*, "A Comparison of Affine Region Detectors," *International Journal of Computer Vision*, pp. 43–72, 2005.

[24] R. Giduthuri and K. Pulli, "OpenVX: A Framework for Accelerating Computer Vision," in *International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 2016, pp. 1–50.

[25] L. Kalms and D. Göhringer, "Exploration of OpenCL for FPGAs using SDAccel and comparison to GPUs and multicore CPUs," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2017. DOI: 10.23919/fpl.2017.8056847.

[26] A. Pieropan, M. Björkman, N. Bergström, and D. Kragic, *Feature descriptors for tracking by detection: A benchmark*, 2016. DOI: 10.48550/ARXIV.1607.06178.

[27] *Cuda akaze github*, "https://github.com/donlk/cuda_akaze", Accessed: 2022-05-19.