An Efficient Description with Halide for IIR Gaussian Filter

Hiroyasu Takagi and Norishige Fukushima
Nagoya Institute of Technology, Japan

***Abstract*—Recursive Gaussian filter is one of the constant-time algorithms fir Gaussian filtering to reduce the computational order. The recursive property makes programs hard for optimization because all resulting pixels in a recursive filter depend on all input pixels. Halide is a domain-specific language for image processing and can powerfully accelerate image processing with a concise description. However, it is also tricky to describe concise scheduling for recursive filters. As a solution to this problem, RecFilter has been developed as a domain-specific language for handling recursive filter, which internally generates the Halide code. The limitation of RecFilter is in approximation accuracy. Here, we extend RecFilter to suit the methods commonly used in recursive Gaussian filter to solve this problem. Also, we improve computational performance by changing computational scheduling. Experimental results showed that the proposed generator produces more computationally efficient codes, and the resulting images have higher approximation accuracy.**

I. Introduction

Approaching the end of Moore's law, computing architecture of chips increases complexity. In the complex hardware, it is hard to maximize the performance of codes. Domain-specific compilers/languages are one of the solutions. Halide [1] is a domain-specific language (DSL) for image processing, and continuously developed [2], [3], [4]. In the Halide, we can separate codes into an algorithm part and a scheduling part to help code optimization. We can write how to work image processing in the algorithm part and how to compute it in the scheduling part. Changing only the scheduling part, we can optimize the code for the specific hardware, such as CPU (x86, ARM, MIPS, Hexagon, PowerPC, and Xeon Phi) and GPU (CUDA, OpenCL, and OpenGL).

Here, we focus on Gaussian filtering in an image processing task in this paper. Gaussian filtering is a smoothing filter used in various signal processing and image processing tasks. For example, Canny edge detection [5], SIFT [6], saliency map computation [7], and image quality metrics of SSIM [8], [9]. Also, edge-preserving filtering, e.g., as bilateral filtering [10], guided image filtering [11] and local Laplacian filter [12], utilizes Gaussian filtering in internal processing for acceleration, such as bilateral filtering [13], [14], [15], [16] guided image filtering [17], and local Laplacian filtering [18].

The computational order of Gaussian filtering depends on the convolution's kernel size, and the kernel size is proportional to σ, which is the standard deviation of the Gaussian distribution. A solution to this problem is a recursive Gaussian filter, which is one of the constant-time algorithms whose computational order is independent of σ. The approximation represents Gaussian filtering by recursive filtering. The representative methods of recursive Gaussian filters are Deriche [19] and Vilet-Young-Verbeek(VYV) [20], [21] forms.

The Halide has difficulty to describe concise scheduling of the recursive filter. Halide's powerful scheduling is tiling; however, the scheduling is prevented by the recursive property that all pixels have a dependency on the other all pixels.

RecFilter [22] has been developed as a DSL for recursive filters to solve this problem. RecFilter wraps Halide and generates Halide code for a recursive filter with a concise description. The limitation of RecFilter is accuracy. The DSL is developed for accelerating recursive filtering, not for generating high accuracy filtering results. The resulting image is not approximated for Gaussian filtering.

Our previous work [23] generates recursive filtering codes with an additional tiling function written in C++ with OpenMP and Halide. This implementation can support tiling for any recursive filtering, but the Halide's main advantage, which is the separability of algorithm and scheduling parts, has disappeared.

In this paper, we propose a new DSL description for recursive filtering, which outputs Halide codes. The DSL resolves the previous works' limitations, such as accuracy in results and modularity in codes. The contributions of this paper are as follows;

- We improve the accuracy of the previous work of RecFilter.
- We extend RecFilter to have representative forms of IIR filtering, such as Deriche and VYV, which have higher accuracy than RecFilter.
- We accelerate the implementation of Deriche and VYV forms by adjusting computational scheduling, which has less dependency than the previous work.

II. Related Works

A. Gaussian filter

A Gaussian filter is a fundamental smoothing filter that weights according to a normal (Gaussian) distribution. The Gaussian kernel has a separability so that a multi-dimensional Gaussian filter can be represented as a product of one-dimensional Gaussian filters. Hence, we will now describe the one-dimensional Gaussian filter.

This work was supported by JSPS KAKENHI 17H01764, 18K19813.

Equation (1) represents the one-dimensional Gaussian kernel;

$$g_n = \tau^{-1}\exp(-\frac{n^2}{2\sigma^2}), \quad \tau = \sum_{n=-R}^{R}\exp(-\frac{n^2}{2\sigma^2}), \quad (1)$$

where σ^2 represents the variance and R is radius. Typically, $R = \lceil 3\sigma \rceil$. Since the computational order of the kernel convolution is $O(R)$, the computational complexity of the Gaussian filter increases in proportion to the σ.

In accelerating Gaussian filtering, there are two types of filters, such as finite impulse response (FIR) filtering and infinite impulse response (IIR) filtering [24]. FIR filters utilize stack of neighboring averaged result [25], [26], [27], [28], [29], which is accelerated by recursive computing. Our previous work [30] accelerated the FIR Gaussian filtering with hard code.

This paper focuses on accelerating IIR Gaussian filtering [19], [21]. IIR filtering supports infinite kernel size. The property is essential for some specific applications, such as interpolating non-uniformly sampled images [31], [32] and hole filling of depth maps by filtering. In both conditions, requiring kernel size is unknown, and it becomes the total image size at worst.

B. Recursive IIR Gaussian Filter

A recursive filter feeds back past output, and is generally represented by the difference equation;

$$y_n = \sum_{i=0}^{N-1} b_i x_{n-i} - \sum_{j=1}^{M} a_j y_{n-j}, \quad (2)$$

where x_n is input signal, y_n is output signal, a_j is feed-back coefficient, b_i is feed-forward coefficient, N and M are the order of the filter.

A recursive Gaussian filter approximates the Gaussian filter by setting the filter's coefficients appropriately. The main approximation methods are Deriche [19] and Vilet-Young-Verbeek (VYV) [20], [21] forms.

The output of the Deriche form is the sum of filtering results in a causal and anti-causal scan. These scans are independent. Also, we need to calculate the initial values for each scan. The initial values are generally computed by directly convoluting extended image edge with the impulse response, which is calculated non-recursive. The image edge extension is generally carried out in a mirroring, and Equation (3) shows the mirror-extended signal $\tilde{f}_n$ from the input signal f_n:

$$\tilde{f}_n = \begin{cases} f_n & (n = 0, ..., N-1) \\ f_{-1-n} & (n < 0) \\ f_{2N-1-n} & (n \geq N). \end{cases} \quad (3)$$

Equation (4) shows the impulse response of the recursive filter:

$$h_n = \sum_{i=0}^{N-1} b_i \delta_{n-i} - \sum_{j=1}^{M} a_j h_{n-j}. \quad (4)$$

Algorithm 1 Algorithm for calculating initial values by convolution with impulse response.

Input: mirror-enhanced input signal $\tilde{f}$, filter coefficients a_i and b_i,accuracy tol
Output: $u_m = h * \tilde{f}$, $(m = 0, ..., M-1)$

compute $h_0, ..., h_M$ with formula 4
$s \leftarrow \sum_{n=0}^{\infty} |h_n|$
$u_m \leftarrow \sum_{n=-m}^{M} h_{n+m}\tilde{f}_{-n}$, $(m = 0, ..., M-1)$
while n=0,1,2,... **do**
  $u_m \leftarrow u_m + h_{n+m}\tilde{f}_{-n}$, $(m = 0, ..., M-1)$
  $s \leftarrow s - |h_n|$
  if $s \leq tol$ **then**
    break
  end if
  $h_{n+M} \leftarrow b_{n+M} - \sum_{j=1}^{M} a_j h_{n+M-1-j}$
end while

```
Func blur_3x3(Buffer<uint8_t> src)
{
    Func clamped, blur_x, blur_y;
    Var x, y, xi, yi, xo, yo;

    // algorithm part
    clamped = BoundaryConditions::repeat_edge(src);
    blur_x(x,y)=(src(x-1, y)
                +(src(x, y)+(src(x+1, y))/3;
    blur_y(x,y)=(blur_x(x-1,y)
                +blur_x(x,y)+blur_x(x+1,y))/3;

    // scheduling part
    blur_y.tile(x, y, xo, yo xi, yi, 32, 32)
          .vectorize(xi, 8).parallel(yo);
    blur_x.compute_at(blur_y, xo).vectorize(xi, 8);

    return blur_y;
}
```

Fig. 1. Halide code of 3×3 box filtering for CPU backend.

Algorithm 1 shows the flow of calculating initial values in the Deriche form.

In the VYV form, the filtering output is obtained by convoluting the result of a causal scan in an anti-causal scan. In this method, the initial values of the causal scan are computed by convolution with impulse response as well as in the Deriche form. In the anti-causal scan, it is calculated by weighting the results of the causal scan [24]. In general, the Deriche form is more computationally expensive than the VYV format, but it is more stable than VYV.

C. Halide

The Halide [1] is a major DSL for image processing. The language is a pure functional language and is embedded in C++. The Halide can be described as a separate description of Algorithm parts and Scheduling parts. Algorithm parts show the essence of processing and are a hardware-independent description. Scheduling parts reveal the computational order and computational method. The former example is scanning-loop order, and the later examples are vectorization and parallelization.

```
RecFilterDim x("x",image_width), y("y",image_height);
RecFilter F("Gaussian");
F.set_clamped_image_border();

// initialize the IIR pipeline
F(x,y) = image(x,y);

// add the filters: causal and anti-causal
F.add_filter(+x, gaussian_weights(sigma, order));
F.add_filter(-x, gaussian_weights(sigma, order));
F.add_filter(+y, gaussian_weights(sigma, order));
F.add_filter(-y, gaussian_weights(sigma, order));

// tile the filter
F.split(x, tile_width);
F.split(y, tile_height);

// schedule the filter
F.set_vectorization_width(vector_width)
F.cpu_auto_schedule();

// JIT compile and run
Buffer<float> out(F.realize());
```

Fig. 2. RecFilter code of recursive Gaussian filter for CPU backend.

Figure 1 shows the Halide code of 3×3 box filtering for CPU backend as an example. *Func* represents a pipeline stage. It is a pure function that defines what value each pixel should have. *Var* is the name to use as variables in the definition of a Func. "*Buffer* src" represents an input image, and its boundaries are extended by a method in the *BoundaryConditions* namespace. "*Var* x,y" show x and y coordinates of images and functions. In the algorithm parts, we horizontally average the clamped image $clamped$, and then vertically mean the averaged image. In scheduling parts, computational scheduling is defined in each func by calling various class methods, e.g., *tile, vectorize, parallel*, and *compute_at*. *tile* split the image into 32×32 tiles by inner and outer variables. *vectorize* orders vectorized computing with SIMD units, e.g., SSE, AVX, and NEON, and this vectorizes pixels along the xi loop. *parallel* shows multi-thread computing with multi-core/thread CPU, and the scheduling parallelize along the yo loop. *compute_at* indicates how to memorize computed results, and we compute and memorize "*Func* blur_x" on xo loop of "*Func* blur_y" under the schedule. In the default schedule, no computation is memorized, i.e., all functions are inlined.

The Halide is utilized for more complex image filtering, such as guided image filtering [33] and weighted median filtering [34].

D. RecFilter

The RecFilter [22] is a DSL for recursive filters with extending the Halide. Figure 2 shows the RecFilter code of recursive Gaussian filter for CPU backend. *RecFilter* is the entity of the recursive filter and *RecFilterDim* is a dimensional variable used by RecFilter. The operator *add_filter* adds one filter at a time in a particular dimension. It specifies the filter dimension in the first argument, the causal/anti-causal direction judging by whether a dimension is positive or negative. The filter coefficients are given in a linear list of the second argument,

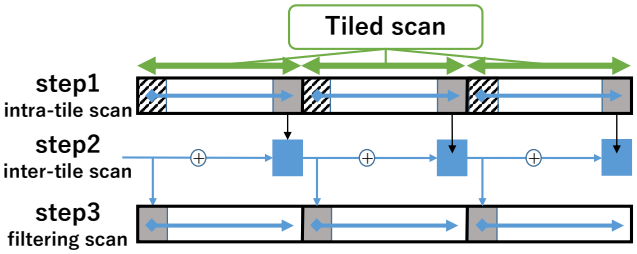


Fig. 3. Filtering method of RecFilter.

whose first element is feed-forward, and remains are feedback coefficients. For tiling, the filter uses the operator *split*. The argument of this gives the dimension to be split and the width of the tiles. Operator *set_vectorization_width* specifies the width of vectorizing, and operator *cpu_auto_schedule* automatically schedules the processing for CPU.

RecFilter accelerates processing by tiling, vectorizing and parallelizing recursive filters. It is necessary to calculate the initial value for each tile for tiling, and this is implemented uniquely in RecFilter. Figure 3 shows a one-directional filtering method of RecFilter. In step 1, the initial value of filtering is padded to zero and then filter on each tile, which is called an intra-tile scan. In step2, first, the results obtained for each tile in step 1 are multiplied by the weights. The weighted results are then added to the adjacent results in the scan direction. This process is called an inter-tile scan. In step 3, the results obtained in step 2 are used as the initial values for filtering, and the final filtering results are obtained. In multi-direction filtering, this filtering will be repeated for multiple dimensions and directions.

RecFilter's automatic scheduler specifies compute_at for intra-tile scans and compute_root for inter-tile scans as calculation timing. This schedule is because the intra-tile scan is closed within each tile, and compute_at improves cache efficiency by sequentially computing each tile. However, since the inter-tile scans are dependent on neighboring tiles, it is necessary to synchronize them, so compute_root is specified.

III. Proposed Method with Experimental Verification

In this paper, we mention the problem that RecFilter has and propose a solution to them. We also extend RecFilter to support methods commonly used methods, such as the Deriche and VYV forms.

A. Solution for limitation in RecFilter

The implementation of RecFilter is similar to the VYV form of the recursive Gaussian filter, where the output of the causal scan is re-filtered in an anti-causal scan to obtain the final output. The filter coefficients are the same as in the VYV form. Note that RecFilter has its way of calculating the filter's initial value, as mentioned in section II-D. This method, in particular, does not retain approximate accuracy when the filter is bi-direction. This limitation is because the intra-scans of RecFilter are not separated in the causal and anti-causal scan,

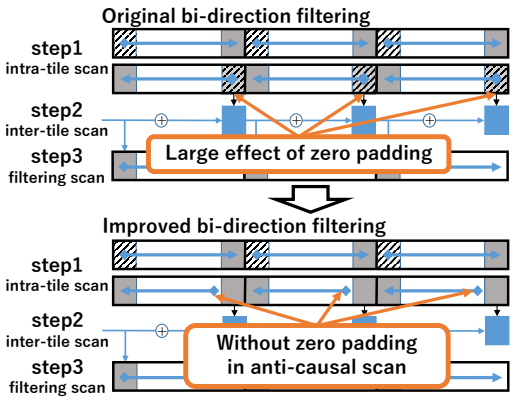


Fig. 4. Improvement of intra-scan.

Table I
SPECIFICATIONS OF COMPUTER.

OS	Windows 10 Enterprise
CPU	Intel(R) Core(TM) i5-4690 @ 3.50GHz
RAM	8GB, DDR3
BUILD	Visual Studio 2017

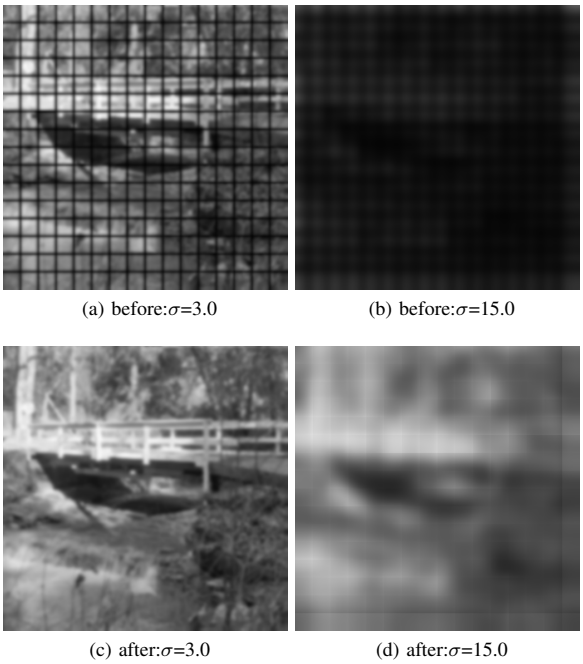
but rather are grouped. Hence, the effect of zero paddings associated with intra-scan in the anti-causal direction is large affected.

We improved this problem by extending RecFilter to not perform zero paddings for the anti-causal direction in intra-scan. Figure 4 illustrates the bi-direction filtering before and after the improvement. In the figure, the shaded areas are padded to zero, and the darker areas at the edge of tiles are used to calculate the initial values. As shown in the figure, the original intra-scan has zero paddings on the areas used to calculate the initial values (the left-side of tiles in the anti-causal scan). This process is why the original RecFilter's approximation accuracy of the initial value is low.

We implemented a recursive Gaussian filter with RecFilter before and after the improvement. Both codes are the same as in Fig. 2. Their outputs with the input image as 512×512 gray image of each are shown in Fig. 5. The specifications of the computer used are also shown in Table I. And, the parameters set in RecFilter are shown in the Table II. Computational time, when scheduled with cpu_auto_schedule of RecFilter, is shown in Fig. 6a. PSNR for outputs of each RecFilter, measured as the output of the FIR Gaussian filter for a correct image, is shown in Fig. 6b. VYV_AVX, used for comparison, is a recursive Gaussian Filter in VYV format that was manually optimized for CPU. From the result of PSNR, the improved RecFilter had high approximation accuracy in a small range of σ, but it decreases significantly with increasing of σ. Furthermore, the computational time does not change significantly either before or after the improvement of RecFilter, and both take several times longer than VYV_AVX. RecFilter requires one extra filtering step for calculating the initial values, and the cost of this process had a considerable effect on the calculation time. We also consider that it is

Table II
PARAMETERS SET IN RECFITER.

order	3
split width	32
vectorize width(x-axis)	8
vectorize width(y-axis)	32



(a) before:σ=3.0 (b) before:σ=15.0

(c) after:σ=3.0 (d) after:σ=15.0

Fig. 5. Outputs of RecFilter before and after improvement.

not easy to achieve high approximation accuracy with this calculation method for initial values.

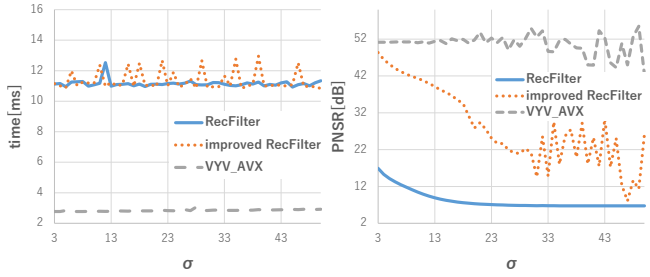
B. Extension of RecFilter Based on VYV Form

The main reason for the low approximation accuracy of RecFilter is in the computing method of initial values for each tile. In this section, we extended RecFilter by improving the computation method for initial values. Also, we adjust the suitable scheduling for the modification. We call this extended RecFilter VYV-RecFilter.

RecFilter originally used filter coefficients based on VYV format; there are two changes in VYV-RecFilter; the way of calculating initial-values and the calculation timing of function.

First, VYV-RecFilter calculates initial values by convolution with impulse response in a causal scan, and then weighting the results of the causal scan in an anti-causal scan [35]. In calculating initial values by convolution with the impulse response, the image edge typically should be extended by mirroring to compensate outside signals. In an inner tile, we can use outside signals of the tile from an input image.

Second, we changed the calculation timing of all functions as compute_at to improve the cache efficiency. In VYV-RecFilter, the processing is closed within a tile, while the



(a) Computational time (b) PSNR

Fig. 6. Execution result of RecFilter before and after improvement.

```
RecFilterDim x("x",image_width), y("y",image_height);
RecFilter F("Gaussian");
F.set_clamped_image_border();

// initialize the IIR pipeline
F(x,y) = image(x,y);

// add the filters: causal and anti-causal
F.add_filter(+x, gaussian_weights(sigma, order));
F.add_filter(-x, gaussian_weights(sigma, order));
F.add_filter(+y, gaussian_weights(sigma, order));
F.add_filter(-y, gaussian_weights(sigma, order));

// specify filtering algorithm
F.algorithm(VYV);
F.set_tol(tol);

// tile the filter
F.split(x, tile_width);
F.split(y, tile_height);

// schedule the filter
F.set_vectorization_width(vector_width)
F.cpu_auto_schedule();

// JIT compile and run
Buffer<float> out(F.realize());
```

Fig. 7. VYV-RecFilter code of recursive Gaussian filter for CPU backend.

original Recfilter has a dependency for each tile.

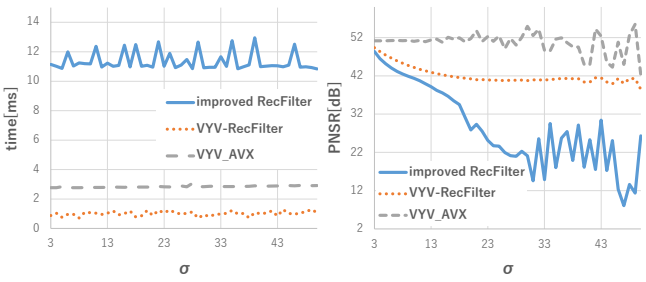
Here, the set parameters are changed to 256 for split width, and 64 for y-axis vectorize width, and otherwise same as before (Table II). Figure 7 shows VYV-RecFilter code of recursive Gaussian filter for CPU backend. We allowed changing the filtering algorithm using the operator *algorithm*. operator *set_tol* also specifies the tolerance of convolution with the impulse response. Outputs of VYV-RecFilter are shown in Fig. 8, and PSNR and computational time are shown in Fig. 9.

The results showed that VYV-RecFilter could significantly accelerate the processing time from RecFilter and improve approximation accuracy. Comparing with VYV_AVX, which is hard C++ code for VYV filter optimized with AVX, however, VYV-RecFilter had lower performance than VYV_AVX.

From the output results (Fig. 8), it can be observed that the boundaries of tiles become noticeable as the increase of σ. This phenomenon is because initial values in the anti-causal scan are calculated only from the result of a causal scan; thus, weakening the dependence of tile boundaries on each other.

(a) σ=3.0 (b) σ=15.0

Fig. 8. Outputs of VYV-RecFilter.



(a) Computational time (b) PSNR

Fig. 9. Execution result of VYV-RecFilter.

C. VYV-RecFilter with Redundancy Calculation

In VYV-RecFilter, the approximation accuracy is degraded because the tile boundaries were not smoothed out due to the calculation method of the initial values in the anti-causal scan. We now provide redundancy in the causal scan to compute the anti-causal scan's initial values at a position far from the tile boundaries, thus smoothing the tile boundaries. We call VYV-RecFilter with redundancy calculation Redundant-VYV-RecFilter, and Fig. 10 shows the filtering method of it. As shown in the figure, we smooth out the tile boundaries by extending the causal scan to outside of the tile and starting the anti-causal scan from there.

Here, set parameters are the same as for VYV-RecFilter. Moreover, the redundancy calculation length is specified 2σ, which was the most accurate. Figure 11 shows Redundant-VYV-RecFilter code of recursive Gaussian filter for CPU backend. Operator *set_redundancy* specifies a redundant amount of calculation.

Figure 12 shows outputs of Redundant-VYV-RecFilter, and Fig. 13 shows result of calculation time and PSNR measurement. The results show that there is not much difference in the computational time between cases with and without the redundancy calculation because the calculation cost of redundant parts is relatively smaller than the other parts. On the other hand, the tile boundaries are smoothed out, and the approximation accuracy is improved. As you can see from the output, the tile boundaries are smoother, even with a large σ than VYV-RecFilter. However, the approximation accuracy

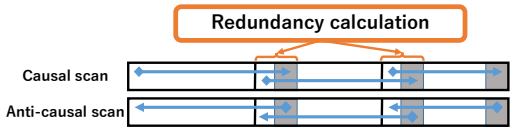


Fig. 10. Filtering method of VYV-RecFilter with Redundancy Calculation.

```
RecFilterDim x("x",image_width), y("y",image_height);
RecFilter F("Gaussian");
F.set_clamped_image_border();

// initialize the IIR pipeline
F(x,y) = image(x,y);

// add the filters: causal and anti-causal
F.add_filter(+x, gaussian_weights(sigma, order));
F.add_filter(-x, gaussian_weights(sigma, order));
F.add_filter(+y, gaussian_weights(sigma, order));
F.add_filter(-y, gaussian_weights(sigma, order));

// specify filtering algorithm
F.algorithm(VYV);
F.set_tol(tol);

// specify the length of redundancy calculation
F.set_redundancy(2*sigma);

// tile the filter
F.split(x, tile_width);
F.split(y, tile_height);

// schedule the filter
F.set_vectorization_width(vector_width)
F.cpu_auto_schedule();

// JIT compile and run
Buffer<float> out(F.realize());
```

Fig. 11. Redundant-VYV-RecFilter code of recursive Gaussian filter for CPU backend.

decreases with increasing σ. This result is due to the low stability of the VYV format.

D. Extension of RecFilter Based on Deriche Form

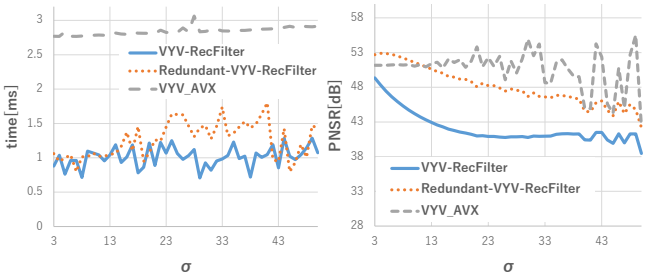
We now extend RecFilter to have Deriche form for further improving the approximation accuracy. We call this extended RecFilter Deriche-RecFilter.

The calculation of initial values in Deriche-RecFilter is the same as a causal scan of VYV-RecFilter, which is a convolution with the impulse response. As for the calculation timing of recursive scanning, compute_at is specified for all functions as well as VYV-RecFilter. The main difference from VYV-RecFilter is anti-causal processing and setting filtering orders. Especially for the anti-causal processing, we also apply convolution processing for initialization.

Here, setting parameters are the same as for VYV-RecFilter. Figure 14 shows Deriche-RecFilter code of recursive Gaussian filter for CPU backend. Unlike the VYV form, the feed-forward coefficients in the Deriche format are not always in order one. Therefore, we overloaded the *add_filter* so that the feed-forward coefficients of any order can be used.

(a) σ=3.0 (b) σ=15.0

Fig. 12. Outputs of Redundant-VYV-RecFilter.



(a) Computational time (b) PSNR

Fig. 13. Execution result of Redundant-VYV-RecFilter.

Figures 15 and 16 show output images, calculation time, and PSNR of Deriche-RecFilter, respectively. Deriche_AVX, used as a comparison, is a recursive Gaussian Filter in Deriche form that was manually optimized for CPU as well as VYV_AVX.

The results show that Deriche-RecFilter is faster than VYV_AVX and Deriche_AVX, and the filter is stable with high approximation accuracy. As you can see from the output (Fig. 15), the processing of tile boundaries, where had a problem with VYV-RecFilter, could be handled smoothly. Deriche-RecFilter also shows higher stability above a certain σ with a similar computational time as Redundant-VYV-RecFilter. We consider that the low approximation accuracy, which was a problem with RecFilter, could be solved.

Figure 17 shows the Halide code generated inside Deriche-RecFilter. For simplicity, only the x-axis filter is described, and the description of the y-axis is omitted. "*Buffer⟨float⟩* iBuffC" and "*Buffer⟨float⟩* iBuffA" in the Fig. 17 are buffers for impulse response in a causal and anti-causal scan. These are pre-computed, and the results are stored in a buffer. Also, ff_c, ff_a, and fb are the filter's coefficients, feed-forward for causal, feed-forward for anti-causal, and feed-back coefficient. Furthermore, the scheduling in Deriche-RecFilter is shown in Fig. 18. Only a causal scan schedule is described for simplicity, but an anti-causal scan is scheduled in the same way. Deriche-RecFilter automatically generates these descriptions and provides a concise description of fast and accurate recursive Gaussian filter.

```
RecFilterDim x("x",image_width), y("y",image_height);
RecFilter F("Gaussian");
F.set_clamped_image_border();

// initialize the IIR pipeline
F(x,y) = image(x,y);

// add the filters: causal and anti-causal
vector<float> coeff_fb,coeff_ff_causal,coeff_ff_anti;
/* calculate coefficents from sigma and order*/
F.add_filter(+x, coeff_ff_causal, coeff_fb);
F.add_filter(-x, coeff_ff_anti, coeff_fb);
F.add_filter(+y, coeff_ff_causal, coeff_fb);
F.add_filter(-y, coeff_ff_anti, coeff_fb);

// specify filtering algorithm
F.algorithm(Deriche);
F.set_tol(tol);

// tile the filter
F.split(x, tile_width);
F.split(y, tile_height);

// schedule the filter
F.set_vectorization_width(vector_width)
F.cpu_auto_schedule();

// JIT compile and run
Buffer<float> out(F.realize());
```

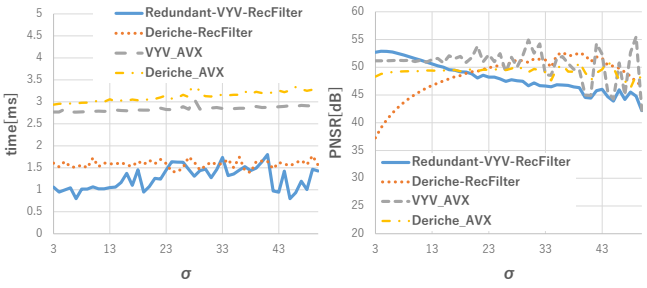
Fig. 14. Deriche-RecFilter code of recursive Gaussian filter for CPU backend.

(a) σ=3.0 (b) σ=15.0

Fig. 15. Outputs of Deriche-RecFilter.

IV. Conclusion

In this paper, we improved the RecFilter, a Halide extension for recursive filtering, to have famous IIR Gaussian filtering styles, such as Deriche and VYV forms. We had changed initialization processing to improves the approximation accuracy. The changed process terminates the dependency between tiles, and then we can also accelerate the filtering time performance. We confirmed that our extension performed better with dozens of code lines than the recursive Gaussian filter, which was optimized with tens times more code.



(a) Computational time (b) PSNR

Fig. 16. Execution result of Deriche-RecFilter.

References

[1] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics*, 31(4):1–12, 2012.

[2] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proc. ACM Programming Language Design and Implementation (PLDI)*, pages 519–530, 2013.

[3] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics*, 35(4):1–11, 2016.

[4] T.-M. Li, M. Gharbi, A. Adams, F. Durand, and J. Ragan-Kelley. Differentiable programming for image processing and deep learning in halide. *ACM Transactions on Graphics*, 37(4):1–13, 2018.

[5] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, 1986.

[6] D. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[7] L. Itti, C. Koch, and E. Niebur. A model of saliency-based visual attention for rapid scene analysis. *IEEE Transactions on Pattern Analysis Machine Intelligence*, 20:1254–1259, 1998.

[8] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[9] T. Sasaki, N. Fukushima, Y. Maeda, K. Sugimoto, and S. Kamata. Constant-time gaussian filtering for acceleration of structure similarity. In *Proc. International Conference on Image Processing and Robot (ICIPRoB)*, 2020.

[10] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *Proc. IEEE International Conference on Computer Vision (ICCV)*, pages 839–846, 1998.

[11] K. He, J. Shun, and X. Tang. Guided image filtering. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 35(6):1397–1409, 2013.

[12] S. Paris, W.S. Hasinoff, and J. Kautz. Local laplacian filters: Edge-aware image processing with a laplacian pyramid. *ACM Trans. on Graphics*, 30(4), 2011.

[13] F. Durand and J. Dorsey. Fast bilateral filtering for the display of high-dynamic-range images. *ACM Trans. on Graphics*, 21(3):257–266, 2002.

[14] K. Sugimoto and S. Kamata. Compressive bilateral filtering. *IEEE Transactions on Image Processing*, 24(11):3357–3369, 2015.

[15] N. Fukushima, K. Sugimoto, and S. Kamata. Complex coefficient representation for iir bilateral filter. In *Proc. International Conference on Image Processing (ICIP)*, 2017.

[16] K. Sugimoto, N. Fukushima, and S. Kamata. 200 fps constant-time bilateral filter using svd and tiling strategy. In *Proc. IEEE International Conference on Image Processing (ICIP)*, 2019.

[17] N. Fukushima, K. Sugimoto, and S. Kamata. Guided image filtering with arbitrary window function. In *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2018.

[18] M. Aubry, S. Paris, J. Kautz, and F. Durand. Fast local laplacian filters: Theory and applications. *ACM Trans. on Graphics*, 33(5), 2014.

[19] R. Deriche. Fast algorithms for low-level vision. *IEEE Transactions on Pattern Analysis Machine Intelligence*, 12:78–87, 1990.

[20] I. T. Young and L. J. vanVliet. Recursive implementation of the gaussian filter. *Signal Processing*, 44:139–151, 1995.

[21] L. J. van Vliet, I. T. Young, and P. W. Verbeek. Recursive gaussian derivative filters. In *Proc. International Conference on Pattern Recognition (ICPR)*, 1998.

[22] G. Chaurasia, J. Ragan-Kelley, S. Paris, G. Drettakis, and F. Durand.

```
Var x;
Var xi, xo;

RDom rxi(0,tile_width);
RDom rk(0, K), rxf(k, tile_width-K);
RDom rxo(0,length/tile_width);

Buffer<float> iBuffC;
/** calc impulse response for causal scan**/
Buffer<float> iBuffA;
/** calc impulse response for anti causal scan**/
RDom rti(0, iBuffC.height());
Expr rx;

/** k is k=0,...,K **/

Func F_init_causal;
rx = tile * xo - rti;
F_init_causal(xi,xo) = input(xo*tile_width+xi);
F_init_causal(rk,xo) += select(rk>=k,
 iBuffC(0,k-1)*input(xo*tile_width+rk-k+1), 0);
F_init_causal(rk, xo) += select(rk == k,
 sum(iBuffC(k, rti)*input(select(rx<0,-rx,rx))),0);

Func F_final_causal;
F_final_causal(xi, xo) = undef;
F_final_causal(rk, xo) = F_init_causal(rx, xo);
F_final_causal(rxf,xo) =
 sum(ff_c(rk)*input(xo*tile_width+rxf-rk, xo))
 +sum(fb(rk)*F_final(rxf-rk-1, xo));

Func F_init_anti_causal;
rx = tile * xo + tile_width - 1 + rti;
F_init_anti_causal(xi, xo) = input(xo*tile_width+xi);
F_init_anti_causal(tile_width-1-rk, xo) += select(
  rk>=k,iBuffA(0,k-1)*input(xo*tile_width-rk+k-1),0);
F_init_anti_causal(tile_width-1-rk, xo) +=
 select(rk == k, sum(iBuffA(k, rti)*input(rx<length,
 rx, 2*length-rx-1,)), 0);

Func F_final_anti_causal;
F_final_anti_causal(xi, xo) = undef;
F_final_anti_causal(tile_width-1-rk, xo) =
 F_init_anti_causal(tile_width-1-rx, xo);
F_final(tile_wifth-1-rxf,xo) =
 sum(ff_a(rk)*
 input(xo*tile_width+tile_width-1-rxf+rk+1,xo))
 +sum(fb(rk)*F_final(tile_width-1-rxf+rk+1,xo));

Func F;        //ouput
F(x) = F_final_causal(x%tile_width, x/tile_width)
 +F_final_anti_causal(x%tile_width, x/tile_width);
```

Fig. 17. Halide code generated inside Deriche-RecFilter.

```
F.compute_root()
  .split(x, xo, xi, split_width)
  .split(y, yo, yi, split_width)
  .reorder(xi, yi, xo, yo)
  .vectorize(xi, vector_width)
  .parallel(xo);
  .parallel(yo);

F_final_causal.compute_at(F, xo)
  .split(y, yo, yi, split_width);
F_final_causal.update(0)
  .split(y, yo, yi, split_width)
  .reorder(rk, yi, xo, yo)
  .vectorize(yi, vector_width)
  .parallel(xo);
  .parallel(yo);
F_final_causal.update(1)
  .split(y, yo, yi, split_width)
  .reorder(rxf, yi, xo, yo)
  .vectorize(yi, vector_width)
  .parallel(xo);
  .parallel(yo);

F_init_causal.compute_at(F_final_causal, xo)
  .split(y, yo, yi, split_width)
  .reorder(rk, yi, xo, yo)
  .vectorize(yi, vector_width)
  .parallel(xo);
  .parallel(yo);
F_init_causal.update(0)
  .split(y, yo, yi, split_width)
  .reorder(rk, yi, xo, yo)
  .vectorize(yi, vector_width)
  .parallel(xo);
  .parallel(yo);
F_init_causal.update(1)
  .split(y, yo, yi, split_width)
  .reorder(rk, yi, xo, yo)
  .vectorize(yi, vector_width)
  .parallel(xo);
  .parallel(yo);
```

Fig. 18. Scheduling of Deriche-RecFilter.

Compiling high performance recursive filters. In *High-Performance Graphics*. ACM Siggraph, 2015.

[23] Y. Tsuji and N. Fukushima. Halide and openmp for generating high-performance recursive filters. In *Proc. International Workshop on Advanced Imaging Technology (IWAIT)*, volume 11515. International Society for Optics and Photonics, 1 2020.

[24] P. Getreuer. A survey of gaussian convolution algorithms. *Image Processing on Line*, 3:286–310, 2013.

[25] E. Elboher and M. Werman. Cosine integral images for fast spatial and range filtering. In *Proc. IEEE International Conference on Image Processing (ICIP)*, 2011.

[26] K. Sugimoto and S. Kamata. Fast gaussian filter with second-order shift property of dct-5. In *Proc. IEEE International Conference on Image Process (ICIP)*, 2013.

[27] K. Sugimoto and S. Kamata. Efficient constant-time gaussian filtering with sliding dct/dst-5 and dual-domain error minimization. *ITE Transactions on Media Technology and Applications*, 3:12–21, 2015.

[28] D. Charalampidis. Recursive implementation of the gaussian filter using truncated cosine functions. *IEEE Transactions on Signal Processing*, 64:3554–3565, 2016.

[29] K. Sugimoto, S. Kyochi, and S. Kamata. Universal approach for dct-based constant-time gaussian filter with moment preservation. *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP2018)*, pages 1498–1502, 2018.

[30] N. Fukushima, Y. Maeda, Y. Kawasaki, M. Nakamura, T. Tsumura, K. Sugimoto, and S. Kamata. Efficient computational scheduling of box and gaussian fir filtering for cpu microarchitecture. In *Proc. Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA), 2018.*, 2018.

[31] L. Lou, P. Nguyen, J. Lawrence, and C. Barnes. Image perforation:automatically accelerating image pipelines by intelligently skipping samples. *ACM Transactions on Graphics*, 35(153), 2016.

[32] N. Fukushima, T. Tsubokawa, and Y. Maeda. Vector addressing for non-sequential sampling in fir image filtering. In *IEEE International Conference on Image Processing (ICIP)*, 2019.

[33] A. Ishikawa, N. Fukushima, A. Maruoka, and T. Iizuka. Halide and genesis for generating domain-specific architecture of guided image filtering. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, 2019.

[34] A. Ishikawa, N. Fukushima, and H. Tajima. Halide implementation of weighted median filter. In *Proc. International Workshop on Advanced Image Technology (IWAIT)*, 2020.

[35] B. Triggs and M. Sdika. Boundary conditions for Young–van Vliet recursive filtering. *IEEE Transactions on Signal Processing*, 54(5):2365–2367, 2006.