IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 18 November 2023, accepted 9 December 2023, date of publication 21 December 2023,
date of current version 18 January 2024.

Digital Object Identifier 10.1109/ACCESS.2023.3345660

RESEARCH ARTICLE

SlidingConv: Domain-Specific Description of Sliding Discrete Cosine Transform Convolution for Halide

YAMATO KANETAKA[1], (Graduate Student Member, IEEE), HIROYASU TAKAGI[2], YOSHIHIRO MAEDA[3], (Member, IEEE), AND NORISHIGE FUKUSHIMA[1], (Member, IEEE)

[1]Department of Engineering, Faculty of Engineering, Nagoya Institute of Technology, Showa-ku, Nagoya 466-8555, Japan
[2]Yamaha Corporation, Hamamatsu, Shizuoka 430-0904, Japan
[3]Department of Electrical Engineering, Faculty of Engineering, Tokyo University of Science, Niijuku, Katsushika-ku, Tokyo 125-8585, Japan

Corresponding author: Norishige Fukushima (fukushima@nitech.ac.jp)

This work was supported in part by Japan Society for the Promotion of Science (JSPS) KAKENHI under Grant 21H03465 and Grant 21K17768, and in part by the Environment Research and Technology Development Fund of the Environmental Restoration and Conservation Agency of Japan under Grant JPMEERF20222M01.

ABSTRACT Filtering is a fundamental tool in image processing, and its acceleration affects many applications. Therefore, various algorithmic and hardware accelerations have been proposed for filtering. Recursive processing using infinite impulse response (IIR) filtering is an efficient algorithm, and various hardware acceleration methods have been applied to IIR filtering. In addition, a domain-specific language (DSL) of RecFilter was proposed to generate efficient IIR code for various hardware applications as an extension of image processing language, Halide. Recursive filters based on sliding discrete cosine transform (SDCT) have been the most efficient approximations in recent years. For hardware acceleration, parallelization of recursive filters is challenging. One of the most efficient methods is tile-based parallelization. However, even if a function is optimized and modularized, it is not sufficiently optimized for applications where various pre/post-processing steps are coupled before and after filtering. Additionally, multiplatform deployment requires reimplementation of the code. In this study, we extended Halide for SDCT convolutions to realize efficient computing of image processing applications with filtering, named SlidingConv. The experimental results showed that SlidingConv is faster than the hand-tuned CPU code and 1/1900 of the hand-tuned code length, running more efficiently than de facto libraries like OpenCV. To verify its efficiency, we deployed the code on various hardware (x86/64 CPU with AVX2/AVX-512, ARM CPU, and GPU). In addition, we verified that the proposed method can accelerate image processing with pre/post-processing for filtering. Our code is available at https://fukushimalab.github.io/SlidingConv/.

INDEX TERMS Image processing, parallel recursive filtering, sliding DCT, domain-specific language, Halide.

I. INTRODUCTION

Spatial filtering is an essential image processing tool in computer graphics, computational photography, and low-level vision applications. Accelerating spatial filters is important because they are used in various applications. The computational cost depends on the filtering window size. A representation of 2D spatial filtering is finite impulse response (FIR) filtering, whose computational order is $\mathcal{O}(R^2)$, where R is the radius of the convolutional kernel. Speeding up the filtering requires algorithmic and hardware acceleration. If a kernel is separable, then the convolution can be computed in $\mathcal{O}(R)$ by decomposing it into two 1D kernels. In addition, the fast Fourier transform (FFT) reduces the convolution order to $\mathcal{O}(\log N)$, where N is the number of signal samples. However, the number of scans of the entire image increases for these accelerations, reducing the cache efficiency (two for separable filtering and three for FFT). These algorithms have

The associate editor coordinating the review of this manuscript and approving it for publication was Andrea F. Abate.

the following optimized libraries: OpenCV, Intel Integrated Performance Primitive (IPP), AMD ROCm Performance Primitives (RPP), NVIDIA Performance Primitives (NPP), FFTW, cuFFT, and clFFT. In addition, a domain-specific language (DSL) [1] for image processing, called Halide [2], can deploy optimized binaries for various hardware. The Halide language can separate descriptions of the algorithm (image processing) and scheduling (loop reordering, parallelization, and vectorization). This separation enables speed improvements and various hardware deployments with a simple description.

Infinite impulse response (IIR) filtering [3], [4], [5], [6], [7], [8] can reduce more orders. The IIR filter is realized by recursive processing, which allows it to be computed at a constant time of $\mathcal{O}(K)$ per pixel, where K is the approximation order ($K \ll R$ in most cases) and is independent of the window size. The 1D IIR filter requires two pass filters, casual and anti-casual directions, and is realized by two IIR systems: parallel and serial systems. The parallel system is the sum of the causal and anticausal results, independently adding forward and reverse filters [3], [4], [5]. The serial system is a product of causal and anticausal results: forward convoluting an input and then reverse convoluting the forward result in a cascade manner [6], [7], [8]. In the 2D or multidimensional signal cases, 1D filters are applied as a cascade 1D filter in a separable manner. The IIR filter was implemented in Insight Toolkit (ITK) [9], developed mainly for medical image processing libraries.

Parallelization of recursive filtering (e.g., IIR) is complex because of its computational dependency; thus, various parallelization approaches have been proposed [10], [11], [12], [13], [14], [15], [16], [17]. A simple parallel execution is a row-by-row or column-by-column parallelization [11], [12]. However, the number of threads in parallelization is limited by the sizes of the rows and columns. In addition, the data I/O efficiency of such parallelization is low owing to the inefficient usage of memory and cache. Therefore, per-tile parallelizing approaches have been proposed to overcome these limitations [13], [14], [15], [16], [17]. These methods use various innovations to initialize the boundary conditions to accommodate dependencies among intertile units.

Tile parallelization code is complex. In addition, convolutions are chained to various additional pre/post-processing steps. A simple example is unsharp masking, which considers the difference between the original image and convolutional image and then adds it to the original image. Furthermore, floating-point operations on 8-bit images require casting as additional pre/post-processing steps. This pre/post-processing reloads the data (i.e., scans the entire image again before and after the filters), reducing the cache efficiency. Loop fusion for pre/post-processing and filtering loops improves cache efficiency; however, implementing loop fusion codes requires rewriting a large amount of code. Moreover, such code-writing cannot be modulated as a function or class, resulting in low portability.

RecFilter [19] was proposed to resolve this modulation issue. RecFilter allows the tile-by-tile parallelization of IIR filters by extending Halide. Recursive filtering requires help with separated descriptions of algorithm and scheduling because Halide mainly targets stencil computing, whereas recursive filtering involves scan computing. Recursive filtering requires initialization processing for each tile, which interleaves the algorithm and scheduling descriptions. RecFilter solves this problem by generating Halide codes. In particular, RecFilter efficiently realizes IIR serial systems where all filters are cascaded with a specific initialization for per-tile parallelization. Moreover, IIR filters can be decomposed into parallel and serial systems to represent the approximate order as a combination of lower-order filters [22], and RecFilter allows constructing filters of appropriate order with parallel systems.

As a recent algorithmic improvement, sliding discrete cosine transform (SDCT) filtering has realized an arbitrary FIR filter as a recursive filter [23], [24], [25], [26], [27], [28], [29]. SDCT can represent FIR filtering as short-time DCTs and compute them as recursive filtering. The computational order is $\mathcal{O}(K)$, as in IIR filters. SDCT filtering can be computed with finite taps instead of an infinite initial length owing to the FIR property. Moreover, the filter can be implemented using a forward filter only (i.e., the number of processes is halved from IIR without an anti-causal filter). Therefore, SDCT filtering requires fewer image scans, and its initialization is easier than IIR filtering. In addition, any even function can be easily designed into a convolution with FIR weights as input, whereas IIR filtering is difficult to design arbitrary weights.

Code complexity in per-tile parallelization for SDCT filtering is inherited from recursive IIR filtering. As an example of increased complexity, the image quality evaluation index, structural similarity (SSIM) [30], which uses an internal Gaussian filter, has been accelerated using an SDCT filter [31], and another [32] which uses a constant-time bilateral filter with SDCT filter and tiling to improve efficiency. However, these applications require considerable code rewriting to realize and deploy applications in various architectures.

Therefore, we propose a domain-specific description for SDCT filtering, named *SlidingConv*. SlidingConv decouples the scheduling and algorithm descriptions for SDCT filtering, generating a high-performance code with fewer descriptions. Fig. 1 shows a visual overview of SlidingConv. SlidingConv generates scheduling-aware Halide codes from pre/post-processing, user-defined kernel, scheduling, and SDCT algorithm descriptions. The generated code is compiled for deployment in various architectures. Halide is not a Turing-complete language; thus, Halide cannot always describe schedules and algorithms separately. SlidingConv solves this contamination problem by acting as a Halide code generator. It also functions as a DSL, inheriting Halide's characteristics as a scheduling descriptive

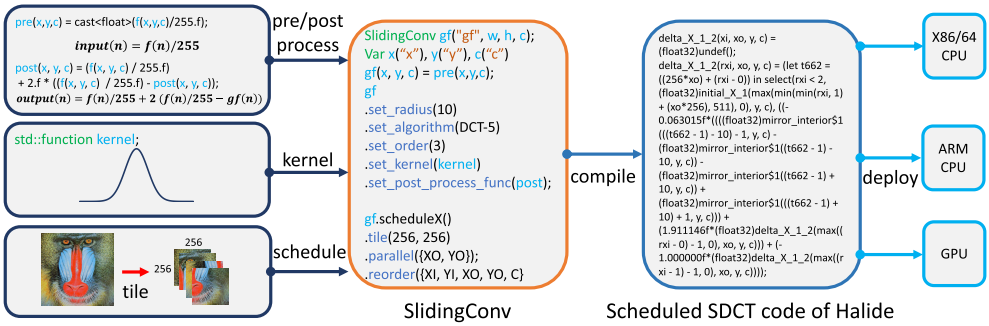


FIGURE 1. Overview of SlidingConv.

TABLE 1. Summary of algorithms and implementations related to convolutional filters (*R*: radius, *N*: image size, *K*: approximation order, $K \ll R$).

algorithm	order	scan	library (CPU)	library (GPU)	DSL
FIR	$\mathcal{O}(\mathrm{R}^2)$	1	OpenCV [18], IPP, RPP	OpenCV [18], NPP, RPP	Halide [2]
FIR-sep	$\mathcal{O}(\mathrm{R})$	2	OpenCV [18], IPP, RPP	OpenCV [18], NPP, RPP	Halide [2]
FFT	$\mathcal{O}(\log \mathrm{N})$	3	IPP, RPP, FFTW	cuFFT, clFFT	Halide [2]
IIR	$\mathcal{O}(\mathrm{K})$	4	ITK [9]	ITK [9], gpufilter [13], [15]–[17]	RecFilter [19], [20]
SDCT	$\mathcal{O}(\mathrm{K})$	2	[21]	N/A	*Proposed

language. Similarly, SlidingConv can be embedded in Halide codes.

This behavior is similar to RecFilter; however, the three problems cannot be solved directly using the conventional RecFilter. First, RecFilter cannot realize recursive filtering in a parallel system. Second, RecFilter initialization is specific to IIR. Third, SDCT filtering uses different data for each approximation order of the parallel system in the recursion, while RecFilter does not require this. Moreover, regarding RecFilter's functionality, only pre-processing can be described with filtering (i.e., no kernel fusion in post-processing [33]).

The contributions of this paper are as follows:

- We propose the first DSL description for SDCT filtering. We support the parallel system of recursive filters, specialize in initialization, and extend the recursive reference data.
- We support loop and kernel fusion for pre and post-processing. The functionality enables cache-efficient descriptions for image processing that use filtering.
- We enable various CPU and GPU backends. SlidingConv is the first implementation of the SDCT filter on GPUs, ARM CPUs, and AVX-512 CPUs. Since SlidingConv outputs Halide codes, it is easily adaptable to various backends but provides unbreakable manual scheduling for the SDCT filter.

Table 1 summarizes the algorithms and implementations of convolutional filters.

II. PRELIMINARIES

A. SLIDING-DCT FILTERING

This section presents the details of SDCT filtering [28]. Here, we present one-dimensional (1D kernel) cases because the cascaded processing of 1D kernels can realize multidimensional kernels with separability. Algorithm 1 provides an overview of SDCT filtering flow.

Algorithm 1 Sliding-DCT-Based Filtering

```
// Calculating first and second terms
for k ← 0 to K do
    Z_k(0) ← G_k Σ_{n=−R}^{+R} C_k(n) f(n)
    Z_k(1) ← G_k Σ_{n=−R}^{+R} C_k(n) f(n + 1)
end for
// Calculating output value for location x = 0, 1
(f ∗ g)(0) ← Σ_{k=0}^{K} Z_k(0)
(f ∗ g)(1) ← Σ_{k=0}^{K} Z_k(1)
// Convoluting cosine terms with a sliding update
for x ← 2 to N − 1 do
    if DCT − I || DCT − V then
        Z_0(x) ← G_0(f(x + R) − f(x − R − 1)) + Z_0(x − 1)
    else if DCT − III || DCT − VII then
        Z_0(x) ← G_0 Δ_k(x−1) + 2C_0(1)Z_0(x−1) − Z_0(x−2)
    end if
    for k ← 1 to K do
        Z_k(x) ← G_k Δ_k(x−1) + 2C_k(1)Z_k(x−1) − Z_k(x−2)
    end for
    // Calculating output value at position x
    (f ∗ g)(x) ← Σ_{k=0}^{K} Z_k(x)
end for
```

1) DEFINITION

Let $f(x) \in \mathcal{I}^W$ $(x \in \mathcal{S} = \{0, 1, \ldots, W-1\} \subset \mathbb{N})$ be the input signals, where $\mathcal{I} = [0.0 : 255.0] \in \mathbb{R}$ is the range domain, W is the size of the signal. Let $g(n) \subset \mathbb{R}^{2R+1}$ be the kernel weight, where $R \in \mathbb{N}$ denotes the kernel radius. The output

TABLE 2. DCT parameters.

DCT-Type	T	k_0	n_0
DCT-I	2R	0	0
DCT-II	2(R + 1)	0	$\frac{1}{2}$
DCT-III	2(R + 1)	$\frac{1}{2}$	0
DCT-IV	2(R + 1)	$\frac{1}{2}$	$\frac{1}{2}$
DCT-V	2(R + 0.5)	0	0
DCT-VI	2(R + 0.5)	0	$\frac{1}{2}$
DCT-VII	2(R + 0.5)	$\frac{1}{2}$	0
DCT-VIII	2(R + 2)	$\frac{1}{2}$	$\frac{1}{2}$

of FIR filtering $(f * g)(x)$ is defined as follows:

$$(f * g)(x) = \sum_{n=-R}^{R} f(x+n)g(n). \tag{1}$$

Here, we consider (1) in the DCT domain. The cosine is an even function; thus, we considered only even functional kernels. The weights $g(n)$ can be represented by the following inverse DCT from the DCT coefficients $G_k \in \mathbb{R}$ and DCT basis $C_k \in \mathbb{R}$ ($k = \{0, 1, \ldots, R\} \subset \mathbb{N}$):

$$g(n) = \sum_{k=0}^{R} G_k C_k(n), \tag{2}$$

where the variables T (or $\phi = \frac{2\pi}{T}$), k_0, and n_0 depend on their type of DCT. DCT can be divided into eight types depending on how the basis, signal, and period are treated as discrete signals for continuous signals, as summarized in Table 2. The coefficients in the matrix form $G = [G_0, G_1, \ldots, G_R]^T \in \mathbb{R}^{1\times R+1}$ can be represented using least squares [28] as follows:

$$G = (C^T W C)^{-1} C^T W g, \tag{3}$$

where $g = [g(0), \ldots, g(R)]^T \in \mathbb{R}^{1\times R+1}$ is the input weight, $C = [C_0, \ldots, C_R]^T \in \mathbb{R}^{R+1\times R+1}$ is the cosine kernels and $W = diag(\frac{1}{2}, 1, \ldots, 1) \in \mathbb{R}^{R+1\times R+1}$ is the weight for dealing with the kernel symmetry.

Convolution (1) can be represented using (2):

$$(f * g)(x) = \sum_{n=-R}^{R}\sum_{k=0}^{R} f(x+n)G_k C_k(n) = \sum_{k=0}^{R} G_k F_k(x), \tag{4}$$

$$F_k(x) = \sum_{n=-R}^{R} f(x+n)C_k(n), \tag{5}$$

where $F_k(x) \in \mathbb{R}$. This expression is $\mathcal{O}(R^2)$ for a 1D convolution whose complexity is higher than that of the original convolution of (1), which is $\mathcal{O}(R)$.

The sliding transform can significantly reduce the order of the increase. This transform utilizes the second-order shift property, which is a relational expression of three short-time transform coefficients for each k: $F_k(x-1)$, $F_k(x)$, and $F_k(x+1)$. The relationship is defined as

$$F_k(x-1) + F_k(x+1) = 2C_k(1-n_0)F_k(x) + \Delta_k(x), \tag{6}$$

TABLE 3. Optimized delta functions for each DCT type.

DCT-Type	optimized $\Delta_k(x)$
DCT-I	$(-1)^k\{f(x-R-1)+f(x+R+1)-C_k(1)(f(x-R)+f(x+R))\}$
DCT-III	$C_k(R)(f(x-R-1)+f(x+R+1))$
DCT-V	$C_k(R)(f(x-R-1)+f(x+R+1)-f(x-R)-f(x+R))$
DCT-VII	$C_k(R)(f(x-R-1)+f(x+R+1)+f(x-R)+f(x+R))$

where the function $\Delta_k(x) \in \mathbb{R}$ is defined by the multiply-add of the input f and coefficients C_k:

$$\begin{aligned}\Delta_k(x) = {} & C_k(-R)f(x-R-1) + C_k(R)f(x+R+1) \\ & -C_k(-R-1)f(x-R) - C_k(R+1)f(x+R).\end{aligned} \tag{7}$$

$\Delta_k(x)$ function can be simplified by extending the terms cos and sin. We can use the following relationships.

- DCT-I: $C_k(\pm R) = (-1)^k$, $C_k(\pm(R+1)) = (-1)^k C_k(1)$
- DCT-III: $C_k(R) = C_k(-R)$, $C_k(\pm(R+1)) = 0$
- DCT-V: $C_k(\pm R) = C_k(\pm(R+1))$
- DCT-VII: $C_k(\pm R) = -C_k(\pm(R+1))$.

Table 3 lists the optimized $\Delta_x^{(k)}$ for each DCT type introduced in [28]. Furthermore, premultiplying $F_k(x)$ by G_k on both sides of (6) simplifies (4).

$$Z_k(x-1) + Z_k(x+1) = 2C_k(1-n_0)Z_k(x) + \Delta_k(x)G_k, \tag{8}$$

where

$$Z_k(x) = G_k F_k(x). \tag{9}$$

The DCT closely approximates the response with fewer coefficients; thus, we can truncate the convolution by a limited value, $K \ll R$. $K \in \mathbb{N}$ is the approximation order. The approximate definition is as follows:

$$(f * g)(x) = \sum_{k=0}^{R} Z_k(x) \simeq \sum_{k=0}^{K} Z_k(x). \tag{10}$$

As Eq. (6) can be updated with two multiplications for each component, the filter of (10) can be approximated by $\mathcal{O}(K)$, which does not depend on the size of the kernel.

We focused on odd types (DCT-I, III, V, and VII) for the usual odd-size filtering in this paper. The even types of DCT-II, IV, VI, and VIII are for an even kernel size (2,4,6,...), used for upsampling and downsampling because the axis is $n_0 = \frac{1}{2}$.

2) DIRECT CURRENT SPECIALIZATION FOR DCT

SDCT can be specialized when $k = 0$ because DCT-I and DCT-V are $k_0 = 0$ and $C_0(n) = \cos(0) = 1$ according to Tab. 2, called a component of direct current (DC).

As $C_0(n) = \cos(0) = 1$, the DC of $F_0(x)$ can be represented as

$$F_0(x) = \sum_{n=-R}^{R} f(x+n) = F_0(x-1) + f(x+R) - f(x-R-1). \quad (11)$$

This relationship is similar to the box filter; thus, it is simple. Thus, (8) can be specialized when $k = 0$ as

$$Z_0(x) = Z_0(x-1) + G_0(f(x+R) - f(x-R-1)). \quad (12)$$

Using (12), $Z_0(x)$ can be computed with fewer operations and higher accuracy than using (8) because there is no round error in floating-point evaluation when multiplying G_k and $C_k(n)$. Furthermore, because K is usually small, this specialization is crucial for speedup factors.

3) OPTIMIZATION OF RADIUS

Given the order K and weight g, it improves filtering accuracy to optimize the convolution radius R for matching the given parameters [26]. This section introduces the optimization approach.

As the FIR filter terminates weights up to a radius R, we should consider the reproduction accuracy of the effective range and ignore the weights of the outer long-tailed portions. The former is the integral of the squared error of the kernel weight $g(t)$ and DCT approximated weight $\tilde{g}(t)$, and this definition is as follows:

$$E_f(K, R) = \eta \int_{-R}^{-R} \{g(t) - \tilde{g}(t)\}^2 \, dt, \quad (13)$$

where

$$\tilde{g}(t) = \sum_{k=0}^{K} G_k C_k(n), \quad \eta = 1/\int_{-\infty}^{\infty} g^2(t)dt. \quad (14)$$

η is the integral of the weight for normalization (e.g., $\sqrt{2\pi}\sigma$ in the Gaussian kernel). The later of the squared errors of the outer kernel function is as follows:

$$E_s(R) = \eta \left\{ \int_{-\infty}^{-(R+1)} g^2(t)dt + \int_{R+1}^{\infty} g^2(t)dt \right\} \quad (15)$$

Then, we minimized the total error to find the optimal R using a linear search, binary search, etc.

$$\arg\min_{R \in W} \left\{ E_s(R) + E_f(K, R) \right\}. \quad (16)$$

This optimization does not affect the filtering speed because it can be calculated before filtering.

B. HALIDE

Halide [2] is a major DSL embedded in C++ for high-performance image processing. It describes the code in the algorithm and scheduling parts separately. The algorithm part includes essential processing and can be described as hardware-independent parts. The scheduling part includes performance tuning of processing and can be described for each hardware architecture. Based on these two parts, a Halide compiler can automatically optimize the code's performance and deploy it to various hardware. Halide is updated continuously and officially [34], [35], [36], [37].

PROGRAM 1. Halide code for 3 × 3 box filtering.

```
1  Func blur_3x3(Buffer<uint8_t> src){
2      Func clamped, blur_x, blur_y;
3      Var x, y, xi, yi, xo, yo;
4      // algorithm part
5      // adding boundary conditions
6      clamped = BoundaryConditions::repeat_edge(src);
7      // horizontal 3 x 1 filtering
8      blur_x(x,y)=(clamped(x−1, y)+clamped(x, y)+
           clamped(x+1, y))/3;
9      // vertical 1 x 3 filtering
10     blur_y(x,y)=(blur_x(x1,y−1)+blur_x(x,y)+blur_x(
           x,y+1))/3;
11     // scheduling part
12     blur_y.tile(x, y, xo, yo xi, yi, 32, 32).vectorize(xi, 8).
           parallel(yo);
13     blur_x.compute_at(blur_y, xo).vectorize(xi, 8);
14     return blur_y;
15 }
```

Program 1 presents an example of Halide 3 × 3 box filtering, and Table 4 lists the main classes and methods for Halide scheduling. The `Func` class is the body of functions, which defines functions using dimensional variables. In the actual pipeline, pure definitions are executed, and updated definitions are executed in the order they are defined. In the `update`, the functions can be defined using reduction domains. In the algorithm part, the input image is padded to prevent external references to the input image, as described by `BoundaryConditions::repeat_edge`, and then separable 3 × 1 filtering is performed. In the scheduling part, `blur_y` is split into 32 × 32 tiles, and variables `x` and `y` are split into inner and outer variables. The inner variable `xi` in `x` is then vectorized by a width of 8, and the outer variable `yo` in `y` is parallelized. `blur_x`'s computation timing is set to `compute_at(blur_y, xo)` that only the necessary pixels are computed and stored within the `xo`-loop of `blur_y`. Vectorization is also performed for `xi` with a width of 8.

C. RECFILTER

RecFilter [19] is a Halide extension for cascaded IIR filtering that internally generates Halide codes to overcome Halide's limitations in recursive filtering. Program 2 shows an example code of RecFilter for IIR Gaussian filtering. `RecFilter` is a class body, `RecFilterDim` is a dimension variable in `RecFilter`. The method `add_filter` adds IIR filtering, which has the first argument for setting the filtering direction and the second for coefficients. For example, `add_filter(+x, {a, b, c})` is the definition of an IIR filter, $F(x) = af(x) + bF(x-1) + cF(x-2)$. The function `gaussian_weights` returns the coefficients for IIR Gaussian Filtering. The `split` method converts a tile dimension into a specific width, similar to Halide scheduling, but

TABLE 4. Major classes and scheduling method of Halide.

Class/Scheduling name	Description
`Func`	Object for containing a function
`Buffer`	Buffer for input, output, and temporary storage for processing result
`Var`	Variable
`RDom`	Reduction domain
`Expr`	An expression consisting of `Func`, `Var`, and `RDom`
realize	Compile and run the setup code
update	Update the definition of `Func` written in algorithm
split	Split a specific dimension into a specific factor
tile	Tile a `Func` by specific dimensions into specific factors
vectorize	Vectorize a specific dimension by a specific width
parallel	Parallelize a specific dimension
reorder	Reorder loops by a specific order
unroll	Unroll a specific loop
compute_at(`Func`, `Var`)	Compute pixels per `Var` loop in `Func`, sequentially
compute_with(`Func`, `Var`)	Make a loop of receiver `Func` and a loop of specific `Func` common
compute_root()	Compute and store all pixels of `Func`
store_at(`Func`, `Var`)	Store pixels per `Var` loop in `Func`
repeat_edge(`Buffer`)	Predefined `Func` for setting a boundary in `BoundaryConditions` namespace

PROGRAM 2. RecFilter code for recursive gaussian filtering.

```
1  RecFilterDim x("x", img_width), y("y", img_height);
2  RecFilter F("Gaussian");
3  // set boundary condition
4  F.set_clamped_image_border();
5  // initialize IIR pipeline by input image
6  F(x,y) = image(x,y);
7  // add recursive filters: causal(+) and
       anticausal(-) x and y directions
8  F.add_filter(+x, gaussian_weights(sigma, order));
9  F.add_filter(−x, gaussian_weights(sigma, order));
10 F.add_filter(+y, gaussian_weights(sigma, order));
11 F.add_filter(−y, gaussian_weights(sigma, order));
12 // scheduling part
13 F.split(x, tile_width);
14 F.split(y, tile_height);
15 F.set_vectorization_width(vector_width)
16 F.cpu_auto_schedule();
17 // JIT compile and run
18 Buffer<float> out(F.realize());
```

specialized for IIR filtering. Then, `cpu_auto_schedule` sets an appropriate schedule with a specific vectorization width from `set_vectorization_width` for RecFilter.

III. LIMITATIONS OF RECFILTER FOR SLIDING-DCT-BASED FILTERING

We clarify the limitations of RecFilter for IIR filtering by trying to realize SDCT filtering of Algorithm 1 by RecFilter. Program 3 presents the pseudo-RecFilter code for the SDCT filter. Lines 4–6 in Program 3 represent lines 3–4 in Algorithm 1, representing the initialized convolutions of the first and second pixels. In addition, Algorithm 1's line 13 or 15 is implemented as a recursive process using RecFilter's `add_filter`, where the array of the second argument contains the feedforward coefficients in the first element and the feedback coefficients in the last elements. The second and third elements are $2C_k(1)$ and -1, respectively, because the second and third terms are $2C_k(1)Z_k(x-1)$ and $-Z_k(x-2)$. The first term is $G_k\Delta_k(x-1)$, which changes depending on the pixel. The first array element is set to 1, and the value of $G_k\Delta_k(x-1)$ is assigned to the expression of RecFilter. Its description is `add_filter(+x, {1, 2 * C1[k], -1})`. In line 10 of Program 3, `select` assigns values equivalent to $G_k\Delta_k(x-1)$ to pixels other than 0 and 1 in RecFilter. However, Program 3 has three problems: 1) loop splitting, 2) no starting point, and 3) fixed-input image problems.

First, RecFilter runs one image scan for each recursive filter. In the image-scanning loop of Algorithm 1's line 11, there are $K+1$ updates for Z_k per pixel, but the processing can be realized by one loop. The existing RecFilter runs the image scanning loop per filter $K+1$ times, added by `add_filter` to the for-loop (i.e., $K+1$ times image scanning loops), such as loop fission. This redundant loop structure causes much of the cache miss. In addition, the filters added by `add_filter` are executed in added order in a cascaded manner, and the second and subsequent filters use the result of the previous filter as an input image. Therefore, convolution using SDCT cannot be realized when multiple filters are processed independently of the input.

Second, RecFilter cannot specify the start pixel of the filter because RecFilter is for IIR filtering, which considers an entire image as an argument. Sliding-DCT-based filtering is an FIR filter that requires a finite-size kernel to be convolved with the pixel. In Algorithm 1's line 2, the first two elements are convolved with the initial values instead of the update processing for the sliding transformation. Whereas IIR filtering does not require the specification of the start pixel, RecFilter does not have the specification functionality.

Third, RecFilter cannot change the input images in the internal RecFilter class. IIR filtering refers to a pixel in the input and the computed images for a newly computed output. SDCT filtering uses the input image in $\Delta_k(x)$ computation, which can be used for input like IIR filtering. However, $\Delta_k(x)$ must be specified for each recursive filter because $\Delta_k(x)$ varies with k in the update process in Algorithm 1's line 13. This requirement interferes with its realization in RecFilter

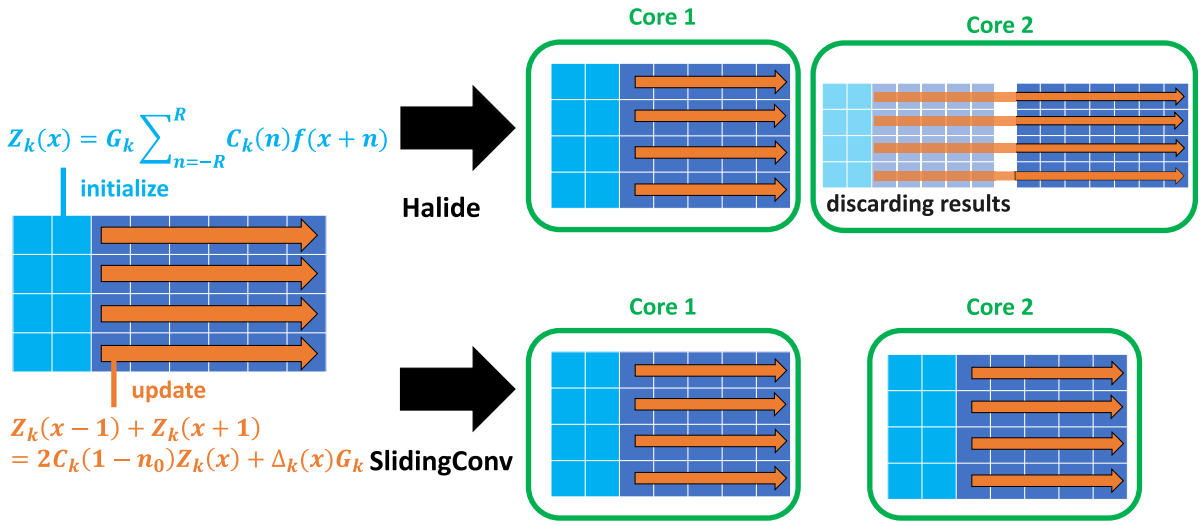


FIGURE 2. The difference between tile scheduling for recursive filtering by Halide and SlidingConv. The first two pixels are initialization calculations, and after that, sliding updates are performed; in the case of Halide, initialization cannot be inserted; thus, the neighboring tiles must also be calculated, while the proposed method completes calculations on a per-tile basis.

PROGRAM 3. Pseudo RecFilter code for sliding transform convolution.

```
1  RecFilter filter_X("filter_X");
2  for (int k = 0; k <= K; k += 1) {
3      // switch first/second term convolution
           and sliding update by delta function
4      Expr convolution = 0.f;
5      // initialized convolution
6      for (int i = -radius; i <= radius; i++)
7          convolution += input(x + i, y, c) * C[i][k] * G[k];
8      // depending on type (See Table 3)
9      Expr delta = compute_delta();
10     filter_X(x, y, c) = select(x == 0 || x == 1,
           convolution, delta)
11     // Zk = 1 * filter(x, y, c) + C_1[k] * Zk(
           x - 1, y, c) -1 * Zl(x - 2, y, c)
12     filter_X.add_filter(+x, {1, 2 * C_1[k], -1});
13 }
```

PROGRAM 4. Internal code for fixing loop splitting problem.

```
1  // Func objects for each order
2  std::vector<Func> filterOrder;
3  for (int k = 0; k <= K; k += 1) {
4      Func filter;
5      // set undef is required for recursive
           description in Halide
6      filter = undef<float>();
7      // definition of recursive filtering for
           each order
8      filter = ...;
9      filterOrder.push_back(filter);
10     // schedule setting
11     if (k != 0) filterOrder[k].update(0).compute_with(
           filterOrder[k-1].update(0), yi);
12 }
```

because RecFilter enforces one image as the filtering input for each cascaded filter.

Because of these three limitations, the existing RecFilter cannot realize SDCT filtering of Algorithm 1.

IV. PROPOSED METHOD

The native Halide code performs massive redundant computations for SDCT filtering with tiling parallelization because each tile requires the results of the previous tile. Therefore, recomputations of the previous tiles are required. The tiling coordinates in the algorithm part must be hardcoded to avoid this condition. SlidingConv can generate such schedule-aware codes using a code that decouples the algorithm from scheduling. Fig. 2 shows the computational scheduling of Halide for SDCT filtering and that of the proposed SlidingConv.

In addition, SlidingConv solves the limitations of RecFilter. Furthermore, SlidingConv allows pre/post-processing of filters to simultaneously, which realizes cache-efficient implementations and provides the functionality of kernel settings in SDCT filters.

A. SOLUTIONS FOR THE RECFILTER'S LIMITATIONS

1) LOOP SPLITTING

For the loop splitting problem, we changed RecFilter to handle parallel systems in a single-loop structure. We newly create as many `Func` objects as the number of filters inside a RecFilter instance to merge the loops using the `compute_with` method, and each `Func` independently refers to the input of RecFilter. The code should specify `update(0)` for scheduling because RecFilter uses `RDom` to create the filter loop. Program 4 shows this modification.

2) NO-STARTING-POINT PROBLEM

To address the no-starting-point problem, we added a method for specifying the initial values necessary to start the update

PROGRAM 5. Internal code for fixing no-starting-point problem.

```
for (int k = 0; k <= K; k += 1) {
    Func filter;
    // feedBackCoeffSize is 2 in SDCT, usually
    int startIndex = feedBackCoeffSize;
    // set undef is required for recursive
        description in Halide
    filter = undef<float>();
    filter(x,y,c)=select(x>=startIndex, initFuncs[k],...);
}
```

PROGRAM 6. Internal code for fixing fixed-input image problem.

```
RecFilter recFilter;
recFilter(x, y, c) = 0;
std::vector<Func> initFuncs, deltaFuncs;
for (int k = 0; k <= K; k += 1) {
    // Calculating first and second terms
    // Appropriate definition for initial
        convolution
    Expr convolution = ...;
    Func initFuncLocal;
    initFunc(x, y, c) = convolution;
    // add an initial function for
        set_init_func
    initFuncs.push_back(initFuncLocal);
    Func deltaLocal;
    // Appropriate definition for delta
    deltaLocal(x, y, c) = ...;
    // add a delta function for set_delta_func
    deltaFuncs.push_back(deltaLocal);
    recFilter.addFilter(+x, {1, 2 * C_1[k], -1});
}
recFilter.set_init_func(initFuncs);
recFilter.set_delta_func(deltaFuncs);
```

process. We added `set_init_func` method to allow this functionality. When the filter added by the `add_filter` method does not have the elements necessary to perform the update process, it refers to the `Func` specified by this method. To specify the initial values for each filter, the same number of `Func` as the number of times `add_filter` passes to `set_init_func`. The first two elements of signals should be specified by the specialized `Func` for initial convolutions, named `initFunc`, allowing RecFilter to start updating the sliding transform from the third element. Inside the internal code of the extended RecFilter (Program 5), the `select` method with the starting point condition switches between the initial value specified by `set_init_func` and the updated definition to define the starting point. Program 6 shows the definition of `initFunc`.

3) FIXED-INPUT IMAGE PROBLEM

We added a method for specifying the reference pixels per filter for the fixed-input image problem. We added the new method, `set_delta_func`, to allow RecFilter to have this functionality. The filter added by `add_filter` refers to the `Func` specified by `set_delta_func`. This allows RecFilter to specify $\Delta_k(x)$ that varies with each k and to refer to a different value for each filter. Program 6 shows the definition of `set_delta_func`.

B. SLIDINGCONV

SlidingConv is implemented in a class in Halide language, defined `SlidingConv`. This is an inherited version of the above improvements to the limitations of RecFilter. SDCT filtering realizes an FIR filter that differs from the IIR filter realized by RecFilter. FIR filters can consider the convolution results more easily than IIR filters; however, they require accurate parameters for the recursion process. SlidingConv also provides transformation and optimization functions for this purpose. Image processing rarely ends with a filter alone but often runs some processes before and after the filter. SlidingConv also allows combining these processes to generate more efficient codes. Table 5 shows the list of methods in SlidingConv. These functionalities are introduced step by step.

Program 7 shows an example code for unsharp masking that uses all the additional functionalities of SlidingConv. The unsharp masking is defined as follows:

$$o = f + m \cdot (f - g * f) = (1 + m) \cdot f - m \cdot g * f. \quad (17)$$

`SlidingConv` is the body for SDCT filtering. SlidingConv's functionalities are described according to Program 7.

1) PRE-PROCESSING

First, we initialized the pixel values in `SlidingConv` by a given `Func` or `Buffer` object. `Func` can contain any image processing; thus, it can write arbitrary pre-processing. The pre-processing is within the loop of the SDCT filtering by default (i.e., no loop splitting). If we plan the pre-processing separately, these processes run individually. In Program 7, pre-processing is converting the data type from `unsigned char` to `float`. No pre-processing is required if we initialize `SlidingConv` by the `SlidingConv` input.

2) ALGORITHM SETTING

Next, we specified an algorithm for `SlidingConv`. The `set_kernel` method sets the kernel weight defined by the `std::function`, and `set_radius` sets its radius. Program 8 is an example of the kernel weight as a Gaussian weight ($\sigma = 3$). However, this std::function does not limit the kernel weight to Gaussian, and an arbitrary even function is available. The weight is described by the type `std::function<double(int, double, int)>`, whose arguments are maxRadius (window radius), offset (n_0 of DCT), and r (radius of required value). If $r = 0$, the code executes the automatic radius determination described in Sec. II-A3. The computational cost of this description does not affect the convolution itself because it is called before filtering, and the cost is lower than that of filtering.

In addition, `set_algorithm` sets the DCT type of SDCT, and `set_order` sets its approximation order. The filtering time and accuracy can also be controlled by specifying the radius, algorithm, and order. These parameters define the characteristics of SDCT filtering.

TABLE 5. Methods of SlidingConv.

method	Description
set_kernel	Set arbitrary kernel weights.
set_order	Set approximation order.
set_radius	Set filtering radius (0 is automatically set optimal value).
set_optimizeCoeff	Set weight optimization.
set_algorithm	Set sliding DCT algorithm DCT-I, III, V, VIII and DCT-I, V without DC optimization.
set_post_process_func	Set post-processing function. Naïve Halide supports pre-processing.
scheduleX	Set scheduling for x-directional filtering.
scheduleY	Set scheduling for x-directional filtering.
cpu_auto_schedule	Set default scheduling for CPU.
gpu_auto_schedule	Set default scheduling for GPU.

PROGRAM 7. SlidingConv code for unsharp masking.

```
1  void unsharpMasking(Buffer<uint8_t> input, int width,
       int height, int channels, std::function kernel)
2  {
3  // variables for x, y, color loops.
4  Var x("x"), y("y"), c("c");
5  SlidingConv gf("gf", width, height, channels);
6
7  // set pre-processing: converting data type
       from unsigned char to float
8  Func pre_process("pre");
9  // input is an 8-bit image
10 pre_process(x, y, c)=cast<float>(input(x, y, c) / 255.f);
11 // if the assignment is input(x,y,c), there
       is no pre-processing.
12 gf(x, y, c) = pre_process(x, y, c);
13 // algorithm part
14 // set kernel weight by std::function object
15 gf.set_kernel(kernel)
16   // set the kernel radius. If 0, then
         automatically set the radius
17   .set_radius(0)
18   // set algorithm of SDCT filtering
19   .set_algorithm(DCT−5)
20   // set order of SDCT filtering
21   .set_order(3);
22 // set post-processing
23 Func post_process("post_process");
24 post_process(x, y, c) = undef<float>();
25 post_process(x, y, c) = (input(x, y, c) / 255.f) + 2.f * ((
       input(x, y, c) / 255.f) − post_process(x, y, c));
26 gf.set_post_process_func(post_process);
27 // Schedule part
28 // for X only
29 gf.scheduleX().tile(width/2,16).parallel({C, XO, YO}).
       reorder({XI,YI,XO,YO,C}).unroll(XI);
30 // for Y only
31 gf.scheduleY().tile(16,height/2).parallel({C, XO, YO}).
       reorder({YI,XI,XO,YO,C}).vectorize(XI);
32 }
```

PROGRAM 8. std::function for gaussian weight kernel.

```
1  double sigma = 3.0;
2  std::function<double(int, double, int)> kernel = [&](int
       maxRadius, double offset, int r){
3      double sum = 0.0; // compute weight summation
4      for (int i = −maxRadius; i <= maxRadius; i++) {
5          double x = i + offset;
6          sum += exp(−(x * x) / (2.0 * sigma * sigma));
7      }
8      double center = r + offset;
9      return exp(center*center/(−2.0*sigma*sigma))/sum;
10 }
```

3) POST-PROCESSING

We can write arbitrary post-processing, running within the SDCT convolution loop (i.e., no loop splitting) with the `set_post_process_func` method. In Program 7, the process corresponding to (17) is performed as a post-processing step. If we do not call the setting function, we can omit post-processing. The writing form is different from that of pre-processing. If `SlidingConv` can accept the `Func` output, we can write the following form, which is similar to pre-processing; $Func(x, y) = SlidingConv(x, y) + 2 * SlidingConv(x, y)$. However, the current implementation has not supported this form due to the limitations of Halide in the language description.

Additional pre/post-processing is limited to point (pixel-wise) operators. Polyhedral optimization must be considered [38], [39] for extending to stencil or areawise operations.

4) SCHEDULING

Finally, we set schedules for `SlidingConv`. SlidingConv has the same functionalities of Halide listed in Table 4. Additionally, we provided two scheduling scope methods: `scheduleX` (for x-direction) and `scheduleY` (for y-direction). We cascaded two SDCT filters in the x-and y-directions; therefore, `SlidingConv` should be scheduled in each direction. We provided the `cpu_auto_schedule` method for default scheduling instead of manual scheduling. In addition, GPU scheduling for the OpenCL or CUDA backend is possible using `gpu_thread` and `gpu_block` `methods` for manual setting, or `gpu_auto_schedule` for the default schedule. The scheduling methods described in Program 7 are equivalent to that in `cpu_auto_schedule`. We provided tile(width/2,16) for X and tile(16,height/2) for Y because they have good speeds for any kernel, order, image size and CPU.

V. EXPERIMENTAL RESULTS

We evaluated SlidingConv based on performance, functionality, and descriptivity and made 9 experiments.

For performance evaluation, the first experiment compared the proposed method with the hund-tuned SIMD code for

sliding DCT algorithms[1] (Sec. V-A1). The second experiment evaluated the filters of various algorithms and implementations based on the computational time and accuracy (Sec. V-A2). The third experiment verified the effectiveness of the proposed method for various architectures by justifying its speed (Sec. V-A3).

For functionality evaluation, we conducted four experiments. The first is the scheduling test for the CPU and GPU (Sec. V-B1), which verifies the parameter of the tiling parallelization. The second is the effectiveness of the pre- and post-filtering features (Sec. V-B2), justifying the performance enhancement of the image-processing chain. The third is the DC optimization (Sec. V-B3), which examines the effectiveness of DC specialization. The fourth is radius optimization (Sec. V-B4), demonstrating the effectiveness of the radius setting.

For descriptivity evaluation, we evaluate two aspects: description efficiency and generated Halide code analysis. We verified the description efficiency (Sec. V-C1) and analyzed SlidingConv's intermediate code using the generated Halide codes (Sec. V-C2) and loop structures (Sec. V-C3).

We set the default schedule as `cpu_auto_schedule` or `gpu_auto_schedule` and the default algorithm as DCT-V. We automatically calculated the radius by setting it to 0 based on (16). Table 6 lists the computers used. CPU1 and GPU1 were the default computers. The codes used are listed in Table 7. The compiler used was Microsoft Visual Studio Professional 2019 for the CPUs, except for the ARM CPU (using gcc).

A. PERFORMANCE

1) COMPARISON WITH HAND-TUNED CODE ON X86/64 CPU

We compared SlidingConv with a hand-tuned AVX SIMD implementation (over 20,000 lines of code) [21] using x86/64 CPUs. We conducted this experiment to show how closely the performance of the proposed method approaches that of the hand-tuned code.

Fig. 3a shows the speed with respect to the approximation order for each type of DCT. SlidingConv was faster than SIMD for all DCT types and orders; however, the difference was smaller from an order of approximately 8. SlidingConv's optimization can accelerate it faster than SIMD, but SlidingConv natively stores more data per order than SIMD. SDCT filtering requires the last two computed results per order for a sliding update of (8), and the older output can be discarded. Ling buffering achieves an effective implementation; however, Halide's specifications do not allow this implementation. Halide only allows the storage of calculation results in loops; it is impossible to keep only the previous result in the loop in the ring buffer. Therefore, SlidingConv, a DSL for Halide language, uses more CPU cache space per order than SIMD and increases the frequency of cache misses with increased orders.

Fig. 3b shows the filtering speed with respect to σ for each DCT type. For all types and orders of DCT, SlidingConv is faster than SIMD but is more affected by σ than SIMD because SlidingConv is more parallelized by tiles. Each tile requires the initial processing of $\mathcal{O}(r)$ convolutions for each order and direction at the start of two pixels; thus, a larger σ (i.e., radius) has a greater impact than smaller SIMD tile cases. The y-scale is quite small, with almost constant-time filtering for each σ.

Fig. 3c presents the approximation accuracy for each order. The accuracy depends on the DCT type, but it is noteworthy that it exceeds 80 dB for approximation order 3. For an 8-bit image, 60 dB is sufficient to match the accuracy of no-approximation [21].

Fig. 4 shows the speed with respect to σ for each image size. SlidingConv is faster than SIMD when the image size is small but slower than SIMD as the image size increases. SlidingConv stores more data per loop because Halide can only store the calculation results per loop unit. SlidingConv uses more cache space per loop than SIMD, and the frequency of cache misses increases with image size.

2) TRADE-OFF BETWEEN APPROXIMATION ACCURACY AND SPEED

We compared the performance evaluation of Gaussian convolution with various algorithms and their implementations. We computed the actual Gaussian convolution responses using the entire image (i.e., for large kernel sizes). Therefore, they are approximated using an IIR or SDCT filter, or the convolution radius is narrowed to a smaller size to speed up the process. Here, we evaluated various implementations regarding the trade-off between speed and approximation accuracy on CPU and GPU. We used the convolution result with a double precision of 6σ as the correct solution and evaluated its approximation accuracy using the peak signal-to-noise ratio (PSNR).

a: CPU

Fig. 5a shows the speed to PSNR with varying approximation orders or radius on Core-i5 10400 CPU. We used 7 methods: SlidingConv, SIMD (AVX) implementation, sepFilter2D—OpenCV's FIR filtering, ITK—class RecursiveGaussianImageFilter in Insight Toolkit (ITK) for IIR filtering, RecFilter [19], RecFilter VYV [20]—an improved RecFilter representation for the Vliet-Young-Verbeek IIR filtering [7], and Halide implementation for separable filtering. SlidingConv is the fastest and has sufficient accuracy because it is in the top-left corner, and SIMD has the nearest performance. The conventional DSL of RecFilter for IIR filtering has low performance (outside the plot) because the implementation is mainly for GPU. The improved RecFilter [20], which is for the CPU, has a high-speed performance; however, its speed is slower than that of SlidingConv, and its accuracy is low.

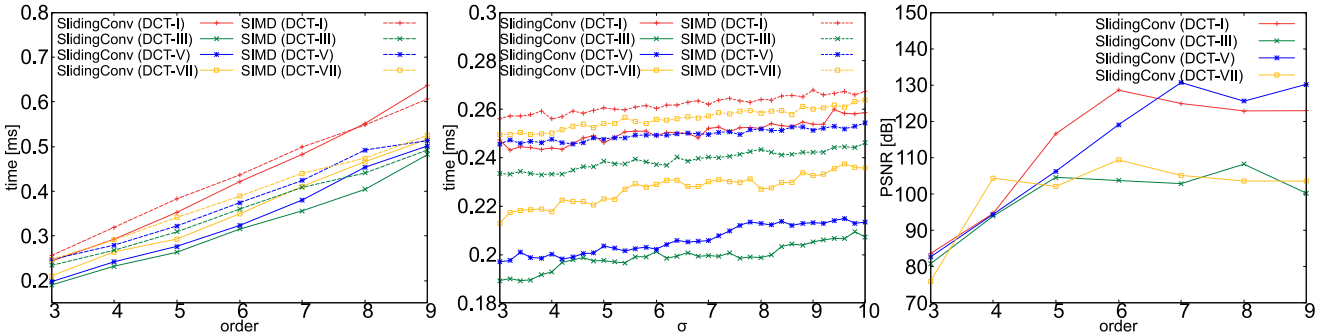
[1] We have no native GPU case because we do not have a hand-tuned GPU code for SDCT. CPU is written in over 20,000 lines; thus, it is hard to rewrite GPU codes, which is one of the motivations for this paper.

TABLE 6. Computational environment. *AVX-512 supported with 2 FMA units. **AVX-512 supported with 1 FMA unit. †ARM CPU (8 P-cores 2 E-cores). CC: CUDA Cores.

Type	Name	No. cores (threads) or No. CC	TFLOPS
CPU1	Intel Core i5-10400	6(12), 2.90-4.30 GHz	0.56-0.83
CPU2*	Intel Core i7-7800X	6(12), 3.50-4.00 GHz	1.34-1.54
CPU3	Intel Core i9-9900K	8(16), 3.60-5.00 GHz	0.92-1.28
CPU4**	Intel Core i9-11980HX	8(16), 3.30-5.00 GHz	0.85-1.28
CPU5*	Intel Core i9-9980XE	18(36), 3.00-4.50 GHz	3.46-5.18
CPU6†	Apple M1 Pro	8+2(10), 0.6-2.06/0.6-3.23 GHz	0.19–0.96
GPU1	NVIDIA GeForce RTX 3060	3584 CC, 1.32-1.78 GHz	9.46-12.74
GPU2	NVIDIA GeForce RTX 2060 Super	2176 CC, 1.47-1.65 GHz	6.40-7.18
GPU3	NVIDIA GeForce RTX 4090	16384 CC, 2.23-2.52 GHz	73.07-82.58

TABLE 7. Used codes for each experiment.

implementation	device	algorithm	Figure/Table
OpenCV+IPP	x86/64 CPU	Sepalable FIR	Figs. 5, 8a
Halide	x86/64 CPU	Sepalable FIR	Fig. 5
ITK	x86/64 CPU	IIR (Deriche)	Fig. 5
RecFilter [19]	x86/64 CPU	IIR	Fig. 5
RecFilter VYV [20]	x86/64 CPU	IIR (VYV)	Fig. 5
C++	x86/64 CPU	Sliding DCT	Fig. 6
C++	ARM CPU	Sliding DCT	Fig. 6
SIMD (AVX)	x86/64 CPU	Sliding DCT	Figs. 3, 4, 5, 6, 8a,Tab. 8
Proposed	x86/64 CPU	Sliding DCT	All
OpenCV+OpenCL	GPU	Sepalable FIR	Fig. 5
Halide	GPU	Sepalable FIR	Fig. 5
ITK	GPU	IIR (VYV)	Fig. 5
RecFilter	GPU	IIR	Fig. 5
gpufilter	GPU	IIR	Fig. 5
Proposed	GPU	Sliding DCT	All



(a) order-time (b) σ-time (c) order-PSNR

FIGURE 3. Computational time of Gaussian filtering for each σ and DCT type on x86/64 CPU. Speed and accuracy of Gaussian filtering for each approximation order and DCT type on x86/64 CPU (Intel Core i5-10400).

These three implementations work more efficiently than the de facto libraries (OpenCV and ITK).

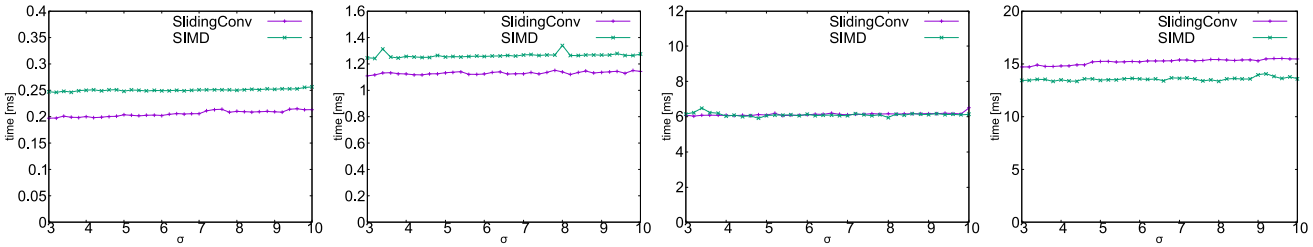
b: GPU

Fig. 5b shows the RTX3060 GPU case. We used 6 implementations: Sliding Conv, gpufilter [15], [17] with alg5f4 option—one of the fastest GPU IIR filtering, OpenCV's sepFilter2D on OpenCL, ITK VYV—ITK's GPU IIR implementation, RecFilter, and Halide implementation for separable filtering. The difference from the CPU experiment is that we changed the plots of SIMD to those of gpufilter and erased those of the improved RecFilter (VYV). Because there was no GPU implementation of the SDCT, we used gpufilter instead. The gpufilter library is a highly efficient convolutional implementation. We omitted the improved RecFilter because it was not created for GPUs.

SlidingConv and gpufilter were faster than the de facto libraries, and SlidingConv was the fastest with sufficient accuracy. The proposed method is faster, and its plots stick to the left edge.

3) DEPLOYMENT ON VARIOUS ARCHITECTURES

This experiment provides an example of the proposed method's effectiveness even when there are insufficient optimization codes for a particular computer architecture.



(a) 512 × 512 (b) 1024 × 1024 (c) 2048 × 2048 (d) 3072 × 3072

FIGURE 4. Computational time of Gaussian filtering for each σ and image size on x86/64 CPU (Intel Core i5-10400).

This is because SlidingConv can optimize codes for various architectures.

a: ARM

The SDCT hand-tuned code is available only for AVX [21]; therefore, the code must be rewritten if the CPU architecture is significantly different. For example, the vector operations must be rewritten for NEON in ARM CPUs. Otherwise, we must use an unoptimized portable C++ code. SlidingConv avoids these problems without changing the code. Fig. 6a shows the filtering speed on M1 Pro (ARM64) and Core i5-10400 (x86/64). We compared the following four implementations: SlidingConv (Intel), SlidingConv (ARM), CPP (Intel), and CPP (ARM). The C++ code of SDCT was not optimized (i.e., without parallelization and vectorization on x86/64).

CPP (ARM) and CPP (Intel) were significantly lower than SlidingConv. SlidingConv (ARM) is as fast as SlidingConv (Intel); SlidingConv is fast regardless of CPU architecture.

b: AVX-512

For the AVX-512 CPU, the same situation occurs with ARM CPUs, which requires the AVX code to be rewritten to the AVX-512 code. Fig. 6b shows the filtering speed on x86/64 CPUs for AVX (i5 10400/i9 9900) and AVX-512 (i7 7800X/i9 11980HX). SIMD is an optimized C++ code with AVX.

SlidingConv is as fast as SIMD on AVX CPUs and faster than SIMD on AVX-512 CPUs. SlidingConv can generate code optimized for AVX-512 through the scheduling method, whereas SIMD only executes AVX instructions. SlidingConv on i7-7800X is approximately 1.9 ms faster and that on i9-11980 is approximately 0.6 ms faster than SIMD. This is the difference in the number of FMA units, with a larger number resulting in more effective optimization.

c: GPU:

In the GPU case, a complete code refresh is usually required, whereas SlidingConv can perform this task only by changing the schedule. Fig. 6c shows the speed of Gaussian filtering on various GPUs: RTX2060 Super (Turing), RTX3060 (Ampere) and RTX4090 (Ada Lovelace).

SlidingConv is fast for all GPU architectures, and the filtering speed is proportional to the GPU performance. The gpufilter had relatively slow results for RTX3060 compared to the others, whereas it had a higher FLOPS than RTX2060 Super. SlidingConv can generate code optimized for various GPU architectures, whereas the gpufilter is optimized for the Turing architecture.

B. FUNCTIONALITY

1) SCHEDULING OF SLIDINGCONV

This section examines the impact of changing the tiling width, which primarily influences SlidingConv scheduling.

a: CPU

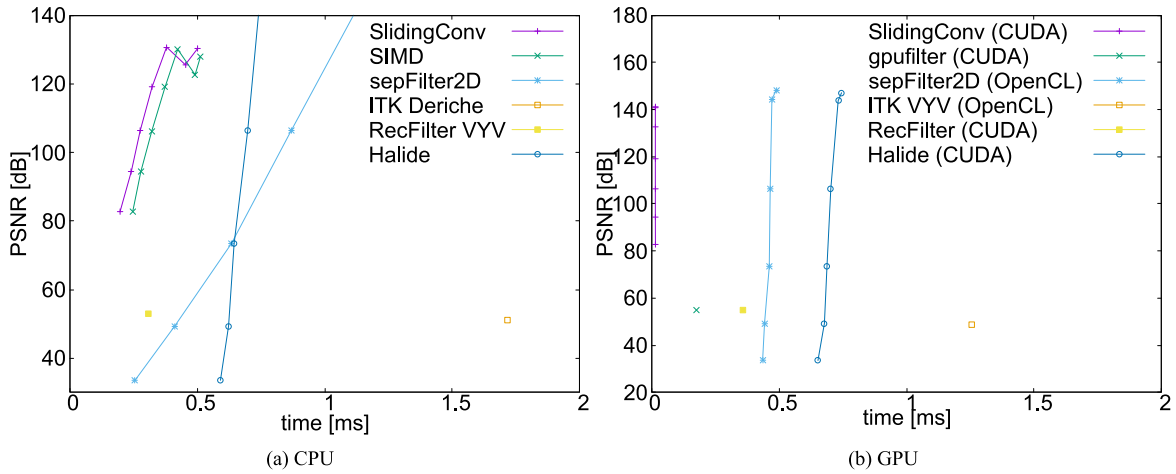
Fig. 7a shows the speed-to-tile width, which can be set using the SlidingConv scheduling method; the other scheduling is the same as cpu_auto_schedule). A tile width 128 was the fastest on i5-10400, whereas 64 was the fastest on i9-9900K and i9-9980XE. This indicated that the optimal tile width depends on the CPUs (clock speed, threads, cache space, etc.) and that different scheduling is required for each CPU to utilize its performance effectively. Therefore, we provide scheduling features for SlidingConv.

b: GPU

Fig. 7b shows various GPUs' results of the filtering speed for changing the tile width scheduling, fixing the other schedule to be the same as gpu_auto_schedule. For RTX2060 Super and RTX3060, a tile width of 32 was the fastest, while for RTX4090, it was 16. This indicated that the optimal tile width also depends on the GPU, and different scheduling is required for each GPU.

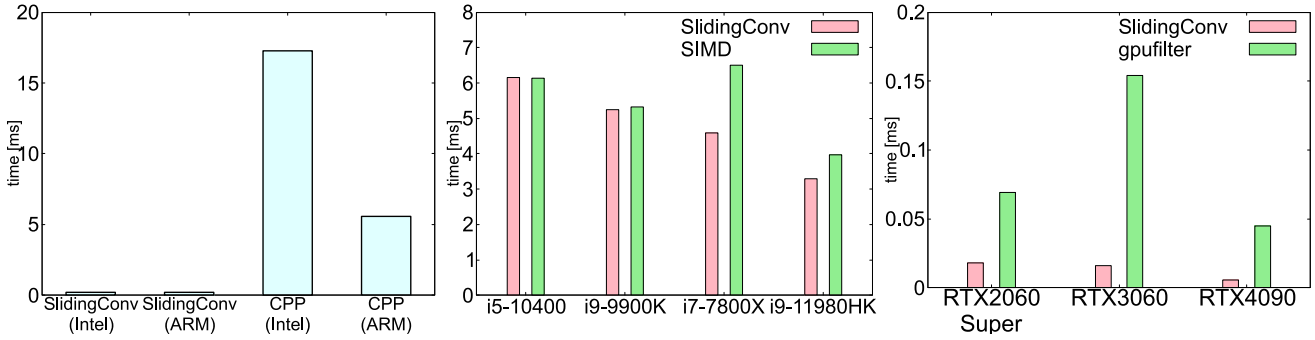
2) PRE/POST-PROCESSING

When adding pre/post-processing to SDCT filtering, the processing must be embedded in the filter code for high efficiency, which requires significant code modification. Fig. 8a shows the speed of unsharp masking, which requires pre-and post-processing. We compared five methods: SlidingConv (Gaussian), Gaussian filtering only by SlidingConv; SlidingConv (Unsharp), Unsharp masking by SlidingConv with pre/post-processing features; SlidingConv (Unsharp



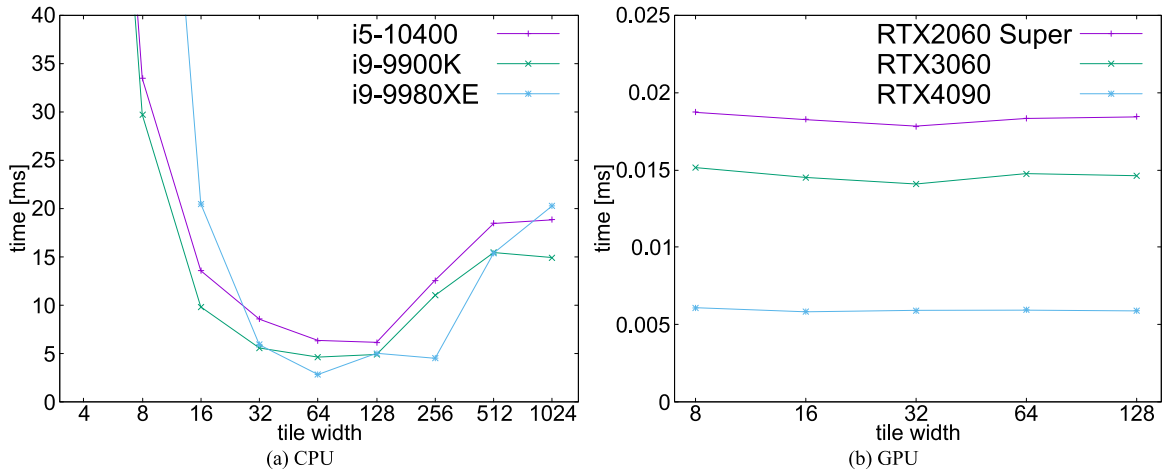
(a) CPU (b) GPU

FIGURE 5. Trade-off between accuracy (PSNR) and speed for various methods on x86/64 CPU (Intel Core i5-10400) and GPU (NVIDIA GeForce RTX 3060).Left-top points indicate high performance. The image size is 512 × 512. The parameters are $\sigma = 3$. Note that the output of RecFilter for CPU is 15.26 ms with 54.90 dB, which is outside of (a).



(a) ARM (b) AVX/AVX-512 (c) GPU

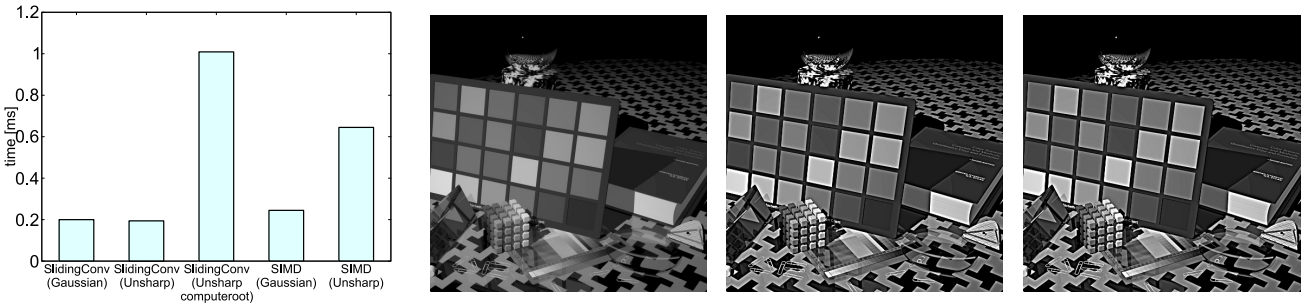
FIGURE 6. Speed of filtering by changing various situations. The image size is 512 × 512 with $\sigma = 3$ for (a), (b), and (d) and 2048 × 2048 with $\sigma = 10$ for (c). The computers used are Intel Core i5-10400 and Apple M1 Pro for (a).



(a) CPU (b) GPU

FIGURE 7. Filtering time for each tile width on x86/64 CPUs and GPUs: 2048 × 2048, $\sigma = 10$, order = 3.

computeroot), Unsharp masking by SlidingConv without pre/post-processing features; SIMD (Gaussian), Gaussian filtering only by SIMD; and SIMD (Unsharp), Unsharp masking by SIMD implementation with a function call chain.



(a) pre/post (b) Original (c) OpenCV output (d) Proposed output

FIGURE 8. The resulting image of unsharp masking ($\sigma = 3.0$, order $= 3$) on Intel Core i5-10400.

SlidingConv (Unsharp) is as fast as SlidingConv (Gaussian); thus, there is little overhead for the pre/post-processing features of SlidingConv. Additionally, SIMD (Unsharp) is significantly lower than SIMD (Gaussian), and SlidingConv (computeroot) is also lower than SlidingConv (Unsharp); it is important to write filtering in a single loop. Therefore, SlidingConv is efficient for writing image processing using SDCT filtering. Figs. 8b, 8c, 8d show the resulting images. The PSNR between the OpenCV and SlidingConv outputs is 76.66 dB, invisible to the human eye.

3) DIRECT CURRENT SPECIALIZATION

This section verifies the effectiveness of the specialization of DC components. Fig. 9 shows the computational time and accuracy of Gaussian filtering for each approximation order with and without DC specialization. DCT-I/V with DC specialization uses the update equaition (12) for $k = 0$ while DCT-I/V without DC uses equaition (8) for $k = 0$. Fig. 9a shows DCT-I/V with DC is approximately one order of magnitude faster than without DC for all DCT types and orders. Therefore, DC specialization is crucial for the speedup factor.

Fig. 9b shows that filtering with DC has higher accuracy than without DC for all DCT and orders because there is no round error in the floating-point arithmetic in (12). Therefore, direct current specialization is crucial for accuracy.

4) OPTIMIZATION OF RADIUS

This section shows the effectiveness of radius size optimization. Fig. 10 shows the accuracy of Gaussian filtering with DCT-V for each approximation order. For $R = \text{opt}$, we optimized the radius using (16). For $R = 6\sigma$, we set the radius to 6σ, sufficient for the typical FIR convolution. The $R = opt$ case has approximately 20 dB higher accuracy than the $R = 6\sigma$ case for all orders. Thus, an appropriate radius setting impacts the filtering accuracy.

C. DESCRIPTIVITY

1) DESCRIPTIVE EFFICIENCY VS SIMD AND HALIDE

We evaluated the efficiency of the code description based on the number of words and code lines. We used the new word counting metrics for the Halide code, and a word counting example is shown in Program 9. Table 8 lists the word counts of SlidingConv and the Halide code generated for the code in Sec. V-A1. We excluded the kernel descriptions from the evaluation.

PROGRAM 9. An example code for word counting.

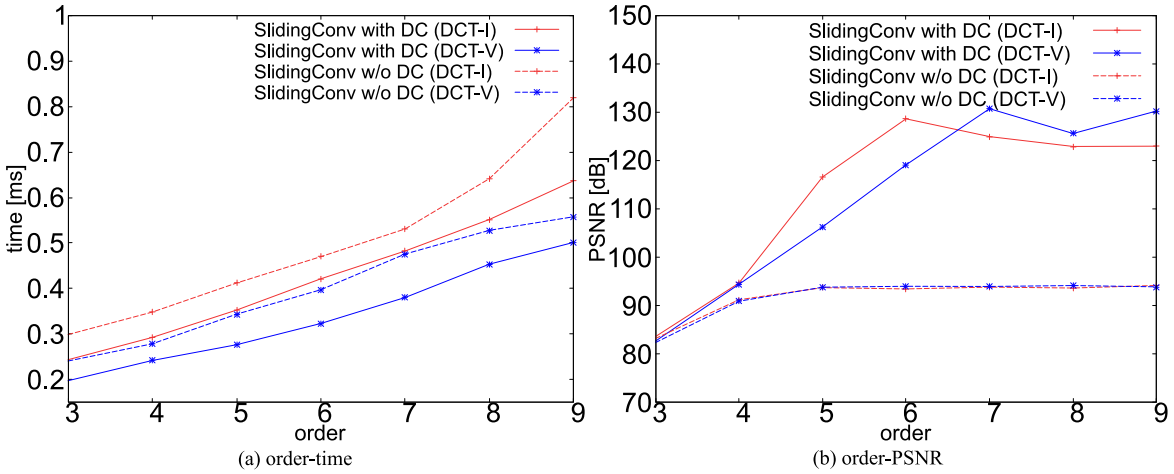
```
1 // 11 words: func, x, y, c, =, 2.f, *, input,
      x, y, c
2 func(x, y, c) = 2.f * input(x, y, c);
3 // 8 words: func, compute_at, root, yi,
      update, 0, vectorize, xi
4 func.compute_at(root, yi).update(0).vectorize(xi);
```

TABLE 8. The number of words and code lines.

	words	lines
SlidingConv	36	11
generated Halide	1352	112
SIMD	-	21229

SlidingConv contains only 36 words, whereas the generated Halide code contains 1352 words (×37 words). Furthermore, the generated code (Program 10, discussed later) includes the scheduling part of `RDom` in the algorithm part. Writing the code is complex, contradicting Halide's advantage of describing the algorithm and scheduling parts separately.

Additionally, we compared the line counts of SlidingConv and manually optimized the SIMD code. Line counting did not include blank lines or comments. SlidingConv has only 11 lines, whereas SIMD has 21229 lines (×1900). SIMD implementation is complex, whereas SlidingConv is simple and highly readable. In addition, if we implement unsharp masking, as shown in Program 7 in SIMD, a tremendous amount of code modification is required; however, we can realize this by adding only a few lines with SlidingConv. Additionally, code optimization can be easily realized through scheduling. We cannot show the SIMD results of loop fusing for unsharp masking because it is difficult to reimplement more than 20,000 lines.



(a) order-time (b) order-PSNR

FIGURE 9. Time and accuracy of Gaussian filtering for each approximation order with $\sigma = 3$ on Intel Core i5-10400.

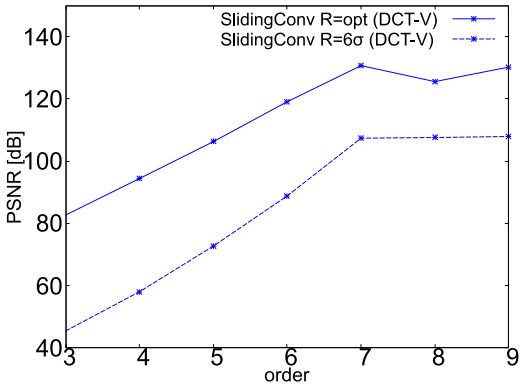


FIGURE 10. Effect of auto radius setting of gaussian filtering for each approximation order. $\sigma = 3$.

2) GENERATED HALIDE CODE

Program 10 shows the generated Halide code of Program 7 (x-direction without scheduling code). The actual volume would be approximately $\times 4$ larger since we have additional codes for scheduling, y-direction, `std::function` kernel settings, etc.

Compared with Program 7, it is more complex because it describes a mixture of scheduling (tiling) and an algorithm (initialization process). This is unavoidable when writing recursive filters in Halide. Therefore, we propose a DSL for SDCT filtering. For example, in Program 10 line 4: xo = (x / 256) and let t636 = ((256*xo) + (rxi − 0)), the algorithm part of the generated Halide code is a scheduling-aware code that considers scheduling by splitting in the filter direction. We introduced a constant value of 256 through scheduling to split the image width. When Halide writes recursive filtering, it requires `RDom` descriptions for the recursion in the algorithm part. To split the recursive filter in this direction, we should change the extent of `RDom`. This description implies that the scheduling of the recursive filter must be considered when writing the algorithm in Halide.

3) GENERATED LOOP STRUCTURE

Program 11 shows the scheduling loop generated for Program 7 (x-direction). In Program 11's lines 24-27, recursive filtering can be processed in a single-pixel loop because the update functions of the delta updating for each $k(\in \{1, 2, 3\})$ are in one loop.

Halide's autoscheduler finds the optimal scheduling from the descriptions of the algorithm-only part; however, Halide cannot find it in recursive filtering because its algorithm part is a scheduling-aware code. For example, when x-directional filtering in a recursive process is vectorized in the x-direction, it is necessary to compute each pixel individually, making it impossible to vectorize them in the filter direction and compute them together, which introduces an incorrect output. This is a limitation of Halide's autoschedulers. Therefore, SlidingConv supports manual scheduling and provides default scheduling via the `cpu_auto_schedule` method.

VI. RELATED WORK

A. PARALLEL RECURSIVE FILTERING

Recursive processing is a typical technique for accelerating filtering. An example is the summed area table (SAT) [40] for box filtering in image processing. In computer vision, this is called an integral image [41]. SAT is a particular case of a first-order recursive filter. SAT can realize two types of constant-time approximations of FIR filters: repeating box filters in serial systems [42], [43] and stacking box filters in parallel systems (stacked integrated image [44], [45]). A survey paper [46] compared accuracy and speed, including IIR filtering; however, this paper focused on a simple C on x86/64 CPUs, which is not optimal from the parallelization and vectorization aspects.

PROGRAM 10. Generated Halide code of Program 7.

```
{
Func SlidingRecFilter_X("SlidingRecFilter_X");
Var x("x"), y("y"), c("c");
SlidingRecFilter_X(x, y, c) = (let
        SlidingRecFilter_X_Final.xo = (x/256) in (let
        SlidingRecFilter_X_Final.xi = (x % 256) in (let
        SlidingRecFilter_X_Final.t727 = float32((uint8)b3
        ((256*SlidingRecFilter_X_Final.xo) +
        SlidingRecFilter_X_Final.xi, y, c)) in ((
        SlidingRecFilter_X_Final.t727/255.000000f) +
        (2.000000f*((SlidingRecFilter_X_Final.t727
        /255.000000f) − ((((float32)deltaF0$1_3(
        SlidingRecFilter_X_Final.xi,
        SlidingRecFilter_X_Final.xo, y, c) + (float32)
        delta_X_1_2(SlidingRecFilter_X_Final.xi,
        SlidingRecFilter_X_Final.xo, y, c)) + (float32)
        delta_X_2_1(SlidingRecFilter_X_Final.xi,
        SlidingRecFilter_X_Final.xo, y, c)) + (float32)
        delta_X_3_0(SlidingRecFilter_X_Final.xi,
        SlidingRecFilter_X_Final.xo, y, c)))))));
}{
Func deltaF0$1_3("deltaF0$1_3");
Var xi("xi"), xo("xo"), y("y"), c("c");
RDom rxi(0, 256);
deltaF0$1_3(xi, xo, y, c) = (float32)undef();
deltaF0$1_3(rxi, xo, y, c) = (let t636 = ((256*xo) + (rxi
        − 0)) in select(rxi < 1, (float32)initial_F0_X(max(
        min(min(rxi, 0) + (xo*256), 511), 0), y, c), (1.0f
        *(0.047619f*((float32)mirror_interior$1(((t636 − 1)
        + 10) + 1, y, c) − (float32)mirror_interior$1((t636
        − 1) − 10, y, c)))) + (1.0f*(float32)deltaF0$1_3(
        max((rxi − 0) − 1, 0), xo, y, c))));
}{
Func delta_X_1_2("delta_X_1_2");
Var xi("xi"), xo("xo"), y("y"), c("c");
RDom rxi(0, 256);
delta_X_1_2(xi, xo, y, c) = (float32)undef();
delta_X_1_2(rxi, xo, y, c) = (let t662 = ((256*xo) + (
        rxi − 0)) in select(rxi < 2, (float32)initial_X_1(
        max(min(min(rxi, 1) + (xo*256), 511), 0), y, c),
        ((−0.063015f*((((float32)mirror_interior$1(((t662 −
        1) − 10) − 1, y, c) − (float32)mirror_interior$1((
        t662 − 1) − 10, y, c)) − (float32)mirror_interior$1
        ((t662 − 1) + 10, y, c)) + (float32)
        mirror_interior$1(((t662 − 1) + 10) + 1, y, c))) +
        (1.911146f*(float32)delta_X_1_2(max((rxi − 0) −
        1, 0), xo, y, c))) + (−1.000000f*(float32)
        delta_X_1_2(max((rxi − 1) − 1, 0), xo, y, c))));
}{
Func delta_X_2_1("delta_X_2_1");
Var xi("xi"), xo("xo"), y("y"), c("c");
RDom rxi(0, 256);
delta_X_2_1(xi, xo, y, c) = (float32)undef();
delta_X_2_1(rxi, xo, y, c) = (let t688 = ((256*xo) + (
        rxi − 0)) in select(rxi < 2, (float32)initial_X_2(
        max(min(min(rxi, 1) + (xo*256), 511), 0), y, c),
        ((0.018142f*((((float32)mirror_interior$1(((t688 −
        1) − 10) − 1, y, c) − (float32)mirror_interior$1((
        t688 − 1) − 10, y, c)) − (float32)mirror_interior$1
        ((t688 − 1) + 10, y, c)) + (float32)
        mirror_interior$1(((t688 − 1) + 10) + 1, y, c))) +
        (1.652478f*(float32)delta_X_2_1(max((rxi − 0) −
        1, 0), xo, y, c))) + (−1.000000f*(float32)
        delta_X_2_1(max((rxi − 1) − 1, 0), xo, y, c))));
}{
Func delta_X_3_0("delta_X_3_0");
Var xi("xi"), xo("xo"), y("y"), c("c");
RDom rxi(0, 256);
```

PROGRAM 10. *(Continued.)* Generated Halide code of Program 7.

```
delta_X_3_0(xi, xo, y, c) = (float32)undef();
delta_X_3_0(rxi, xo, y, c) = (let t714 = ((256*xo) + (
        rxi − 0)) in select(rxi < 2, (float32)initial_X_3(
        max(min(min(rxi, 1) + (xo*256), 511), 0), y, c),
        ((−0.002311f*((((float32)mirror_interior$1(((t714 −
        1) − 10) − 1, y, c) − (float32)mirror_interior$1((
        t714 − 1) − 10, y, c)) − (float32)mirror_interior$1
        ((t714 − 1) + 10, y, c)) + (float32)
        mirror_interior$1(((t714 − 1) + 10) + 1, y, c))) +
        (1.246980f*(float32)delta_X_3_0(max((rxi − 0) −
        1, 0), xo, y, c))) + (−1.000000f*(float32)
        delta_X_3_0(max((rxi − 1) − 1, 0), xo, y, c))));
}
```

PROGRAM 11. Generated scheduling loop of Program 7.

```
produce SlidingRecFilter_X:
  parallel c in [0, 2]: //color-loop
    parallel y.yo in [0, 31]: //image-y-loop
      parallel x.xo in [0, 1]: //image-x-loop
        for y.yi in [0, 15]: //tile-y-loop
          produce initial_X_3:
            produce initial_X_2:
              produce initial_X_1:
                produce initial_F0_X:
                  vectorized fused.x: //
                      convolution for
                      initilize Z_k values
                    initial_X_3(...) = ...
                    initial_X_2(...) = ...
                    initial_X_1(...) = ...
                    initial_F0_X(...) = ...
          consume initial_X_3:
            consume initial_X_2:
              consume initial_X_1:
                consume initial_F0_X:
                  produce delta_X_3_0:
                    produce delta_X_2_1:
                      produce delta_X_1_2:
                        produce deltaF0:
                          unrolled rxi in [0, 255]:
                            delta_X_3_0(...) = ...
                                //order 3
                            delta_X_2_1(...) = ...
                                //order 2
                            delta_X_1_2(...) = ...
                                //order 1
                            deltaF0(...) = ... //DC
                  consume delta_X_3_0:
                    consume delta_X_2_1:
                      consume delta_X_1_2:
                        consume deltaF0:
                          for x.xi in [0, 255]: //
                              tile-x-loop
                            SlidingRecFilter_X(...)
                                = ...
```

The core of the recursive processing of prefix sums, $y_i = x_i + y_{i-1}$, is not easy for parallelization; thus, there are various implementations of recursive filters on GPUs. First, a massively parallel GPU implementation

parallelizes the rows and columns without dependencies [47]. However, this is not sufficiently parallel for massively parallel environments, and the data locality is low. Therefore, parallelization by tiling has been considered for a long time [10], and the GPU implementation of the recursive filter by Nehab et al. used the superposition property to reduce the dependence on tile computation and improve efficiency [13]. This filter is only available in serial type. Other serial-type filters have been implemented in the Insight Toolkit (ITK) [9] with highly efficient CPU and GPU implementations [14]. As an extension, there are also implementations of additive forms, such as Deriche's form [3], which are more highly parallel [15]. These IIR filters require the consideration of infinite-length taps; however, there are efficient implementations [16]. More efficient implementations extend to GPUs for 3D filtering [17]. These implementations [13], [15], [16], [17] are summarized on the following page.[2]

B. SHORT-TIME FOURIER AND SLIDING TRANSFORMS

The fast Fourier transform [48] is a classical acceleration technique, and various libraries provide its functions (FFTW [49], Intel MKL, cuFFT). Recently, the short-time Fourier transform (STFT) [50] has been used for local analysis. Libraries for STFT included MATLAB stft, PyTortch pythorch.stft, TensorFlow tf.signal.stft, librosa [51], and the large time-frequency analysis toolbox (LTFAT) [52]. STFT is slower than FFT in usual calculations because STFT computes FFT redundant on a block-by-block basis. However, a sliding transform reduces the order and is called a sliding discrete Fourier transform (SDFT) or a sliding window discrete Fourier transform (SWDFT). The sliding transform efficiently computes frequencies using an incremental formula based on the relationship between adjacent frequency components in a short-time DFT/DCT/DST. SDFT is summarized in detail in the following papers [53], [54], [55].

Frequency transformations have many applications, including linear convolutions, correlations, and spectral analyses. Among these, SDCT is used for convolution, and specialized transforms for convolution have been proposed. Convolutions usually require a forward transform, a filter, and an inverse transform; short-time transforms also require the same chains. However, convolution-specific methods hide the inverse transform in the flow, and SDCT filters are one such method. Initially, SDCT filters were used in the context of integral images [56] but were refined to various types of SDCT filters: I [23], [24], [29], III [27], V [25], [26], and VII [28].

Regular FIR filters have several parallelization patterns [57] and can be implemented efficiently depending on the situation. However, there is still little research on the efficient hardware implementation of SDCT filters, which are FIR filters realized as recursive filters. This research is limited to CPU [21], [58]. The 1-pass 2D filter [58] is highly cache-efficient with fewer synchronizations, although only multichannel data can be vectorized. This is useful for the multichannel-friendly filter [32], [59] and constant-time stereo matching [60]. An efficient vectorization method [21] using FMA has also been proposed for pure SDCT filters.

C. DSL FOR IMAGE PROCESSING

In parallel computing, various computing patterns exist [61], such as map, stencil, reduction, and scan patterns. In addition, parallel computation and memory structures vary depending on computer architecture. Therefore, programmers must program carefully according to solving problems and use the appropriate language depending on the architecture (e.g., NVIDIA CUDA, AMD HIP, HSL, OpenCL, OpenMP, OpenACC, MPI, SYCL).

DSLs that optimize the data stream to compute these patterns efficiently have been proposed in various domains. Stencil computation is a representative field of parallel DSL [62], [63], [64] because stencil patterns are used in various scientific problems. Image processing is a popular field in DSLs. Image processing involves the most parallel patterns; however, most image-processing algorithms include map, stencil, and reduction patterns.

A representative DSL for image processing is Halide language [2],[3] and the development of Halide is still active until 2023. Image-processing DSLs typically optimize the stencil computations and stream programs. HIPAcc [65][4] is similar to Halide, and Forma [66][5] has autoscheduling functionality. Currently, Halide also has mature autoscheduling [34], [35], [36], [37], [67], [68], which has been verified in various applications [69], [70]. PolyMage [38][6] uses the polyhedral model [71] for loop parallelization, which is more flexible for complex loop structures. These DSLs are used in multicore CPUs and GPUs.

More specific DSLs exist in image processing for FPGAs. Darkroom [72][7] is an early image-processing DSL for FPGA that was embedded in Terra to extract the maximum reuse of data. SPIRAL [73][8] is a DSL limited to signal processing. The following research extends functionalities, such as Rigel [74],[9] RIPL [75],[10] SODA [76],[11] and Aetherling [77].[12] In addition, extensions of image-processing DSL to include an FPGA backend were proposed. PolyMage's FPGA expansion [78], HIPACC-FPGA [79], [80],[13] and HipaccVX [81][14] have been proposed. Halide has been

[2]https://github.com/andmax/gpufilter

[3]https://halide-lang.org/
[4]http://hipacc-lang.org/
[5]https://github.com/NVIDIA/Forma
[6]https://github.com/bollu/polymage
[7]http://darkroom-lang.org/
[8]https://spiral.net/index.html
[9]https://github.com/jameshegarty/rigel
[10]https://github.com/robstewart57/ripl
[11]https://github.com/UCLA-VAST/soda
[12]https://aetherling.org/
[13]https://github.com/hipacc/hipacc-fpga
[14]https://github.com/HipaccVX/HipaccVX

extended to include an FPGA backend, such as Halide-HLS [82][15] Halide to FPGAs [83] and Hetero-Halide [84].[16] Halide has also been extended to other computing units such as DSPs [85], push memory [86] and TensorCore [87]. To support accelerators while considering computation scheduling functions, DSLs for directly generating native language code (e.g., C and C++) rather than extending the DSL are also emerging, such as Exo Language [88].[17]

Current DSLs are extended to more specialized and narrow domains (e.g., machine learning) and broader domains. For deep learning DSL, TVM [89], [90],[18] Tensor comprehensions [91][19] and Ansor [92] have also been proposed. TVM shares many similarities with Halide modified from the Halide IR. DSLs for deep learning are reviewed in [93]. For more general-purpose DSLs, a polyhedral model for loop parallelization was used for wider domain adaptation. A polyhedral compiler can be used for image processing but is not limited to image processing: PLUTO [94],[20] Polly [95],[21] PENCIL [96],[22] and TIRAMISU [97].[23] These DSLs can be used for Halide, and Distributed Halide [98] is using Polly for better scheduling. Further research has been proposed to convert binary files and old native code to DSL. Helium [99][24] can convert compiled x86 binary codes into Halide codes, and DEXTER [100][25] can automatically translate image-processing libraries into Halide.

Another extension direction is to move away from the stencil calculations; one is the proposed method. SlidingConv was used for the scan pattern with the tiling strategy instead of using stencil patterns for the usual convolution. RecFiler [19] and its extension [20] are similar extension patterns. Most image-processing DSLs are not Turing complete, and all Halide computations are over regular grids. Thus, recursions must have bounded domains, and additional loop generation by scheduling, such as initialization in SlidingConv, was not allowed. Indigo [101][26] and Opt [102][27] support image processing in matrix representation for inverse image processing, which commonly uses matrix inverse and FFT. RandConv extends Halide's stencil computations to sparse stencil computations [103].

VII. CONCLUSION

We present a new DSL for image convolution based on a sliding DCT for high-performance computing on various architectures. The sliding DCT can accelerate convolution's fundamental image processing tools by converting FIR filtering to recursive filtering. However, recursive processing prevents efficient parallel image processing and complicates the code. We solved this problem by providing only a scheduling interface and hiding the complex high-efficiency parallel code generation within the DSL. In addition, our DSL supports optimal code generation with minimal code modification in various situations, adding pre/post-processing for filtering and changing architectures (x86/64 AVX, AVX-512 CPU, ARM CPU, and GPU). We show that our DSL works more efficiently than the de facto libraries of OpenCV and ITK and the conventional work of RecFilter [19] and gpufilter [15], [17] on various CPU and GPU architectures. We also show the description efficiency that our DSL has equivalent or better performance than hand-tuned CPU-implemented code with a 1/1900 code length.

[15] https://github.com/jingpu/Halide-HLS
[16] https://github.com/UCLA-VAST/heterohalide
[17] https://exo-lang.dev/
[18] https://tvm.apache.org/
[19] https://facebookresearch.github.io/TensorComprehensions/
[20] https://github.com/bondhugula/pluto
[21] https://polly.llvm.org/
[22] https://github.com/pencil-language
[23] https://tiramisu-compiler.org/
[24] http://projects.csail.mit.edu/helium/
[25] https://dexter.uwplse.org/
[26] https://mbdriscoll.github.io/indigo/
[27] http://optlang.org/

REFERENCES

[1] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, Dec. 2005.
[2] 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 Trans. Graph.*, vol. 31, no. 4, pp. 1–12, Aug. 2012.
[3] R. Deriche, "Fast algorithms for low-level vision," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 12, no. 1, pp. 78–87, Jan. 1990.
[4] R. Deriche, "Recursively implementating the Gaussian and its derivatives," INRIA, Nat. Center Sci. Res. (CNRS), Paris France, Res. Rep. RR-1893, 1993, p. 24. [Online]. Available: https://inria.hal.science/inria-00074778 and https://about.hal.science/en/legal-notice/
[5] G. Farnebäck and C.-F. Westin, "Improving deriche-style recursive Gaussian filters," *J. Math. Imag. Vis.*, vol. 26, no. 3, pp. 293–299, Dec. 2006.
[6] I. T. Young and L. J. van Vliet, "Recursive implementation of the Gaussian filter," *Signal Process.*, vol. 44, no. 2, pp. 139–151, Jun. 1995.
[7] L. J. van Vliet, I. T. Young, and P. W. Verbeek, "Recursive Gaussian derivative filters," in *Proc. Int. Conf. Pattern Recognit. (ICPR)*, 1998, pp. 509–514.
[8] B. Triggs and M. Sdika, "Boundary conditions for young-van Vliet recursive filtering," *IEEE Trans. Signal Process.*, vol. 54, no. 6, pp. 2365–2367, Jun. 2006.
[9] T. Yoo, M. Ackerman, W. Lorensen, W. Schroeder, V. Chalana, S. Aylward, D. Metaxas, and R. Whitaker, "Engineering and algorithm design for an image processing API: A technical report on ITK-the insight toolkit," Med. Meets Virtual Reality 02/10, Nat. Library Med., Bethesda, MD, USA, Tech. Rep., 2002, pp. 586–592.
[10] W. Sung and S. K. Mitra, "Efficient multi-processor implementation of recursive digital filters," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process. (ICASSP)*, vol. 11, Apr. 1986, pp. 257–260.
[11] X. Wang and B. E. Shi, "GPU implemention of fast Gabor filters," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2010, pp. 373–376.
[12] D. Ruijters and P. Thévenaz, "GPU prefilter for accurate cubic B-spline interpolation," *Comput. J.*, vol. 55, no. 1, pp. 15–20, Jan. 2012.
[13] D. Nehab, A. Maximo, R. S. Lima, and H. Hoppe, "GPU-efficient recursive filtering and summed-area tables," *ACM Trans. Graph.*, vol. 30, no. 6, pp. 1–12, Dec. 2011.
[14] I. Vidal-Migallon, O. Commowick, X. Pennec, J. Dauguet, and T. Vercauteren, "GPU and CPU implementation of Young–Van Vliet's recursive Gaussian smoothing filter," *Insight J.*, p. 16, Jul. 2013.
[15] A. Maximo, "Efficient finite impulse response filters in massively-parallel recursive systems," *J. Real-Time Image Process.*, vol. 12, no. 3, pp. 603–611, Oct. 2016.
[16] D. Nehab and A. Maximo, "Parallel recursive filtering of infinite input extensions," *ACM Trans. Graph.*, vol. 35, no. 6, pp. 1–13, Nov. 2016.
[17] A. Maximo, "GPU efficient 1D and 3D recursive filtering," *Digit. Signal Process.*, vol. 114, Jul. 2021, Art. no. 103076.

[18] G. Bradski, "The OpenCV library," *Dr. Dobb's J. Softw. Tools*, vol. 25, no. 11, pp. 120–123, 2000.

[19] G. Chaurasia, J. Ragan-Kelley, S. Paris, G. Drettakis, and F. Durand, "Compiling high performance recursive filters," in *Proc. 7th Conf. High-Perform. Graph. (HPG)*, Aug. 2015, pp. 85–94.

[20] H. Takagi and N. Fukushima, "An efficient description with halide for iir Gaussian filter," in *Proc. Asia–Pacific Signal Inf. Process. Assoc. Annu. Summit Conf. (APSIPA ASC)*, 2020, pp. 28–35. [Online]. Available: https://ieeexplore.ieee.org/document/9306460

[21] T. Otsuka, N. Fukushima, Y. Maeda, K. Sugimoto, and S.-I. Kamata, "Optimization of sliding-DCT based Gaussian filtering for hardware accelerator," in *Proc. IEEE Int. Conf. Vis. Commun. Image Process. (VCIP)*, Dec. 2020, pp. 423–426.

[22] L. Alvarez and L. Mazorra, "Signal and image restoration using shock filters and anisotropic diffusion," *SIAM J. Numer. Anal.*, vol. 31, no. 2, pp. 590–605, Apr. 1994.

[23] E. Elboher and M. Werman, "Cosine integral images for fast spatial and range filtering," in *Proc. 18th IEEE Int. Conf. Image Process. (ICIP)*, Sep. 2011, pp. 89–92.

[24] K. Sugimoto and S.-I. Kamata, "Fast image filtering by DCT-based kernel decomposition and sequential sum update," in *Proc. 19th IEEE Int. Conf. Image Process. (ICIP)*, Sep. 2012, pp. 125–128.

[25] K. Sugimoto and S.-I. Kamata, "Fast Gaussian filter with second-order shift property of DCT-5," in *Proc. IEEE Int. Conf. Image Process.*, Sep. 2013, pp. 514–518.

[26] K. Sugimoto and S.-I. Kamata, "[Paper] efficient constant-time Gaussian filtering with sliding DCT/DST-5 and dual-domain error minimization," *ITE Trans. Media Technol. Appl.*, vol. 3, no. 1, pp. 12–21, 2015.

[27] D. Charalampidis, "Recursive implementation of the Gaussian filter using truncated cosine functions," *IEEE Trans. Signal Process.*, vol. 64, no. 14, pp. 3554–3565, Jul. 2016.

[28] K. Sugimoto, S. Kyochi, and S.-I. Kamata, "Universal approach for DCT-based constant-time Gaussian filter with moment preservation," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Apr. 2018, pp. 1498–1502.

[29] T. Yano, K. Sugimoto, Y. Kuroki, and S.-I. Kamata, "Acceleration of Gaussian filter with short window length using DCT-1," in *Proc. Asia–Pacific Signal Inf. Process. Assoc. Annu. Summit Conf. (APSIPA ASC)*, Nov. 2018, pp. 129–132.

[30] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: From error visibility to structural similarity," *IEEE Trans. Image Process.*, vol. 13, no. 4, pp. 600–612, Apr. 2004.

[31] T. Sasaki, N. Fukushima, Y. Maeda, K. Sugimoto, and S.-I. Kamata, "Constant-time Gaussian filtering for acceleration of structure similarity," in *Proc. Int. Conf. Image Process. Robot. (ICIP)*, Mar. 2020, pp. 1–6.

[32] K. Sugimotoy, N. Fukushimazy, and S.-i. Kamatay, "200 FPS constant-time bilateral filter using SVD and tiling strategy," in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, Sep. 2019, pp. 190–194.

[33] G. Wang, Y. Lin, and W. Yi, "Kernel fusion: An effective method for better power efficiency on multithreaded GPU," in *Proc. IEEE/ACM Int. Conf. Green Comput. Commun. Int. Conf. Cyber, Phys. Social Comput.*, Dec. 2010, pp. 344–350.

[34] 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 SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, vol. 48, no. 6, 2013, pp. 519–530.

[35] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, "Automatically scheduling halide image processing pipelines," *ACM Trans. Graph.*, vol. 35, no. 4, pp. 1–11, Jul. 2016.

[36] T.-M. Li, M. Gharbi, A. Adams, F. Durand, and J. Ragan-Kelley, "Differentiable programming for image processing and deep learning in halide," *ACM Trans. Graph.*, vol. 37, no. 4, pp. 1–13, Aug. 2018.

[37] A. Adams, K. Ma, L. Anderson, R. Baghdadi, T.-M. Li, M. Gharbi, B. Steiner, S. Johnson, K. Fatahalian, F. Durand, and J. Ragan-Kelley, "Learning to optimize halide with tree search and random programs," *ACM Trans. Graph.*, vol. 38, no. 4, pp. 1–12, Aug. 2019.

[38] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "PolyMage: Automatic optimization for image processing pipelines," in *Proc. Int. Conf. Archit. Support Program. Lang. Operating Syst. (ASPLOS)*, 2015, pp. 429–443.

[39] J. Zhao and A. Cohen, "Flextended tiles: A flexible extension of overlapped tiles for polyhedral compilation," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, p. 47, 2019.

[40] F. C. Crow, "Summed-area tables for texture mapping," in *Proc. 11th Annu. Conf. Comput. Graph. Interact. Techn.*, 1984, vol. 18, no. 3, pp. 207–212.

[41] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Dec. 2001, pp. 511–518.

[42] W. M. Wells, "Efficient synthesis of Gaussian filters by cascaded uniform filters," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. PAMI-8, no. 2, pp. 234–239, Mar. 1986.

[43] P. Gwosdek, S. Grewenig, A. Bruhn, and J. Weickert, "Theoretical foundations of Gaussian convolution by extended box filtering," in *Scale Space and Variational Methods in Computer Vision* (Lecture Notes in Computer Science), Berlin, Germany: Springer, 2012, pp. 447–458.

[44] A. Bhatia, W. E. Snyder, and G. Bilbro, "Stacked integral image," in *Proc. IEEE Int. Conf. Robot. Autom.*, May 2010, pp. 1530–1535.

[45] E. Elboher and M. Werman, "Efficient and accurate Gaussian image filtering using running sums," in *Proc. 12th Int. Conf. Intell. Syst. Design Appl. (ISDA)*, Nov. 2012, pp. 897–902.

[46] P. Getreuer, "A survey of Gaussian convolution algorithms," *Image Process. Line*, vol. 3, pp. 286–310, Dec. 2013.

[47] J. Hensley, T. Scheuermann, G. Coombe, M. Singh, and A. Lastra, "Fast summed-area table generation and its applications," *Comput. Graph. Forum*, vol. 24, no. 3, pp. 547–555, Sep. 2005.

[48] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.*, vol. 19, no. 90, pp. 297–301, 1965.

[49] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proc. IEEE*, vol. 93, no. 2, pp. 216–231, Feb. 2005.

[50] J. B. Allen and L. R. Rabiner, "A unified approach to short-time Fourier analysis and synthesis," *Proc. IEEE*, vol. 65, no. 11, pp. 1558–1564, Nov. 1977.

[51] B. McFee, C. Raffel, D. Liang, D. P. Ellis, M. McVicar, E. Battenberg, and O. Nieto, "Librosa: Audio and music signal analysis in Python," in *Proc. Python Sci. Conf. (SciPy)*, vol. 8, 2015, pp. 18–25.

[52] Z. Průša, P. L. Søndergaard, N. Holighaus, C. Wiesmeyr, and P. Balazs, "The large time-frequency analysis toolbox 2.0," in *Proc. 10th Int. Symp. Comput. Music Multidisciplinary Res.* Marseille, France: Springer, Oct. 2013, pp. 419–442.

[53] E. Jacobsen and R. Lyons, "The sliding DFT," *IEEE Signal Process. Mag.*, vol. 20, no. 2, pp. 74–80, Mar. 2003.

[54] E. Jacobsen and R. Lyons, "An update to the sliding DFT," *IEEE Signal Process. Mag.*, vol. 21, no. 1, pp. 110–111, Jan. 2004.

[55] R. Lyons and C. Howard, "Improvements to the sliding discrete Fourier transform algorithm [tips & tricks]," *IEEE Signal Process. Mag.*, vol. 38, no. 4, pp. 119–127, Jul. 2021.

[56] M. Hussein, F. Porikli, and L. Davis, "Kernel integral images: A framework for fast non-uniform filtering," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2008.

[57] Y. Maeda, N. Fukushima, and H. Matsuo, "Taxonomy of vectorization patterns of programming for FIR image filters using kernel subsampling and new one," *Appl. Sci.*, vol. 8, no. 8, p. 1235, Jul. 2018.

[58] N. Fukushima, Y. Maeda, Y. Kawasaki, M. Nakamura, T. Tsumura, K. Sugimoto, and S.-I. Kamata, "Efficient computational scheduling of box and Gaussian FIR filtering for CPU microarchitecture," in *Proc. Asia–Pacific Signal Inf. Process. Assoc. Annu. Summit Conf. (APSIPA ASC)*, Nov. 2018, pp. 875–879.

[59] K. Mishiba, "Fast guided median filter," *IEEE Trans. Image Process.*, vol. 32, pp. 737–749, 2023.

[60] Z. Ma, K. He, Y. Wei, J. Sun, and E. Wu, "Constant time weighted median filtering for stereo matching and beyond," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Dec. 2013, pp. 49–56.

[61] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*. Amsterdam, The Netherlands: Elsevier, 2012.

[62] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 235–244, Jun. 2007.

[63] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey, "3.5-D blocking optimization for stencil computations on modern CPUs and GPUs," in *Proc. ACM/IEEE Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2010, pp. 1–13.

[64] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The pochoir stencil compiler," in *Proc. 23rd Annu. ACM Symp. Parallelism Algorithms Architectures*, Jun. 2011, pp. 117–128.

[65] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, and W. Eckert, "HIPAcc: A domain-specific language and compiler for image processing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 1, pp. 210–224, Jan. 2016.

[66] M. Ravishankar, J. Holewinski, and V. Grover, "Forma: A DSL for image processing applications to target GPUs and multi-core CPUs," in *Proc. 8th Workshop Gen. Purpose Process. Using GPUs*, Feb. 2015, pp. 109–120.

[67] S. Sioutas, S. Stuijk, L. Waeijen, T. Basten, H. Corporaal, and L. Somers, "Schedule synthesis for halide pipelines through reuse analysis," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 2, pp. 1–22, Jun. 2019.

[68] S. Sioutas, S. Stuijk, T. Basten, H. Corporaal, and L. Somers, "Schedule synthesis for halide pipelines on GPUs," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 3, pp. 1–25, Sep. 2020.

[69] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina, C. Kozyrakis, and M. Horowitz, "Interstellar: Using Halide's scheduling language to analyze DNN accelerators," in *Proc. 25th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, Mar. 2020, pp. 369–383.

[70] H. Nogami, S. Oishi, T. Sasaki, Y. Maeda, and N. Fukushima, "Performance evaluation of halide auto-scheduler with directional cubic convolution interpolation," in *Proc. Int. Workshop Adv. Imag. Technol. (IWAIT)*, vol. 12592, Mar. 2023, Art. no. 125922.

[71] C. Lengauer, "Loop parallelization in the polytope model," in *Proc. Int. Conf. Concurrency Theory (CONCUR)*, 1993, pp. 398–416.

[72] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, "Darkroom: Compiling high-level image processing code into hardware pipelines," *ACM Trans. Graph.*, vol. 33, no. 4, pp. 1–144, 2014.

[73] P. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Computer generation of hardware for linear digital signal processing transforms," *ACM Trans. Design Autom. Electron. Syst.*, vol. 17, no. 2, pp. 1–33, Apr. 2012.

[74] J. Hegarty, R. Daly, Z. DeVito, J. Ragan-Kelley, M. Horowitz, and P. Hanrahan, "Rigel: Flexible multi-rate image processing hardware," *ACM Trans. Graph.*, vol. 35, no. 4, pp. 1–11, Jul. 2016.

[75] R. Stewart, K. Duncan, G. Michaelson, P. Garcia, D. Bhowmik, and A. Wallace, "RIPL: A parallel image processing language for FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 11, no. 1, pp. 1–24, Mar. 2018.

[76] Y. Chi, J. Cong, P. Wei, and P. Zhou, "SODA: Stencil with optimized dataflow architecture," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2018, pp. 1–8.

[77] D. Durst, M. Feldman, D. Huff, D. Akeley, R. Daly, G. L. Bernstein, M. Patrignani, K. Fatahalian, and P. Hanrahan, "Type-directed scheduling of streaming accelerators," in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, Jun. 2020, pp. 408–422.

[78] N. Chugh, V. Vasista, S. Purini, and U. Bondhugula, "A DSL compiler for accelerating image processing pipelines on FPGAs," in *Proc. Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, Sep. 2016, pp. 327–338.

[79] O. Reiche, M. Schmid, F. Hannig, R. Membarth, and J. Teich, "Code generation from a domain-specific language for C-based HLS of hardware accelerators," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synth. (CODE+ISSS)*, Oct. 2014, pp. 1–10.

[80] O. Reiche, M. A. Özkan, R. Membarth, J. Teicha, and F. Hannig, "Generating FPGA-based image processing accelerators with hipacc," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2017, pp. 1026–1033.

[81] M. A. Özkan, B. Ok, B. Qiao, J. Teich, and F. Hannig, "HipaccVX: Wedding of OpenVX and DSL-based code generation," *J. Real-Time Image Process.*, vol. 18, no. 3, pp. 765–777, Jun. 2021.

[82] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, "Programming heterogeneous systems from an image processing DSL," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 3, pp. 1–25, Sep. 2017.

[83] A. Ishikawa, N. Fukushima, A. Maruoka, and T. Iizuka, "Halide and GENESIS for generating domain-specific architecture of guided image filtering," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2019, pp. 1–5.

[84] J. Li, Y. Chi, and J. Cong, "HeteroHalide: From image processing DSL to efficient FPGA acceleration," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, Feb. 2020, pp. 51–57.

[85] S. Vocke, H. Corporaal, R. Jordans, R. Corvino, and R. Nas, "Extending halide to improve software development for imaging DSPs," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 3, p. 21, 2017.

[86] Q. Liu, J. Setter, D. Huff, M. Strange, K. Feng, M. Horowitz, P. Raina, and F. Kjolstad, "Unified buffer: Compiling image processing and machine learning applications to push-memory accelerators," *ACM Trans. Archit. Code Optim.*, vol. 20, no. 2, pp. 1–26, Jun. 2023.

[87] S. Sioutas, S. Stuijk, T. Basten, L. Somers, and H. Corporaal, "Programming tensor cores from an image processing DSL," in *Proc. 23th Int. Workshop Softw. Compil. Embedded Syst.*, May 2020, pp. 36–41.

[88] Y. Ikarashi, G. L. Bernstein, A. Reinking, H. Genc, and J. Ragan-Kelley, "Exocompilation for productive programming of hardware accelerators," in *Proc. 43rd ACM SIGPLAN Int. Conf. Program. Lang. Design Implement.*, Jun. 2022, pp. 703–718.

[89] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proc. USENIX Conf. Oper. Syst. Design Implement. (OSDI)*, 2018, pp. 579–594. [Online]. Available: https://dl.acm.org/doi/10.5555/3291168.3291211

[90] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," in *Proc. Neural Inf. Process. Syst. (NIPS)*, 2018, pp. 3393–3404. [Online]. Available: https://dl.acm.org/doi/10.5555/3327144.3327258

[91] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," 2018, *arXiv:1802.04730*.

[92] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J. E. Gonzalez, and I. Stoica, "Ansor: Generating high-performance tensor programs for deep learning," in *Proc. USENIX Conf. Oper. Syst. Design Implement. (OSDI)*, 2020, p. 49. [Online]. Available: https://dl.acm.org/doi/abs/10.5555/3488766.3488815

[93] M. Li, Y. Liu, X. Liu, Q. Sun, X. You, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian, "The deep learning compiler: A comprehensive survey," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 3, pp. 708–727, Mar. 2021.

[94] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral program optimization system," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2008, pp. 101–113.

[95] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly—Performing polyhedral optimizations on a low-level intermediate representation," *Parallel Process. Lett.*, vol. 22, no. 4, Dec. 2012, Art. no. 1250010.

[96] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. Van Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev, "PENCIL: A platform-neutral compute intermediate language for accelerator programming," in *Proc. Int. Conf. Parallel Archit. Compilation (PACT)*, Oct. 2015, pp. 138–149.

[97] R. Baghdadi, J. Ray, M. B. Romdhane, E. D. Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, Feb. 2019, pp. 193–205.

[98] T. Denniston, S. Kamil, and S. Amarasinghe, "Distributed halide," *ACM SIGPLAN Notices*, vol. 51, no. 8, pp. 1–12, Nov. 2016.

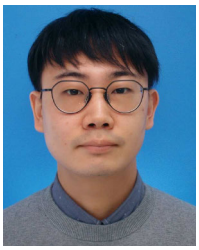
[99] C. Mendis, J. Bosboom, K. Wu, S. Kamil, J. Ragan-Kelley, S. Paris, Q. Zhao, and S. Amarasinghe, "Helium: Lifting high-performance stencil kernels from stripped x86 binaries to halide DSL code," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2015, pp. 391–402. [Online]. Available: https://dl.acm.org/doi/10.1145/2737924.2737974

[100] M. B. S. Ahmad, J. Ragan-Kelley, A. Cheung, and S. Kamil, "Automatically translating image processing libraries to halide," *ACM Trans. Graph.*, vol. 38, no. 6, pp. 1–13, Dec. 2019.

[101] M. Driscoll, B. Brock, F. Ong, J. Tamir, H.-Y. Liu, M. Lustig, A. Fox, and K. Yelick, "Indigo: A domain-specific language for fast, portable image reconstruction," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2018, pp. 495–504.

[102] Z. Devito, M. Mara, M. Zollhöfer, G. Bernstein, J. Ragan-Kelley, C. Theobalt, P. Hanrahan, M. Fisher, and M. Niessner, "Opt: A domain specific language for non-linear least squares optimization in graphics and imaging," *ACM Trans. Graph.*, vol. 36, no. 5, pp. 1–27, Oct. 2017.
[103] H. Takagi and N. Fukushima, "Domain specific description in halide for randomized image convolution," in *Proc. Asia–Pacific Signal Inf. Process. Assoc. Annu. Summit Conf. (APSIPA ASC)*, Dec. 2021, pp. 63–69. [Online]. Available: https://ieeexplore.ieee.org/document/9689317

YAMATO KANETAKA (Graduate Student Member, IEEE) received the B.E. degree from the Nagoya Institute of Technology, Japan, in 2022, where he is currently pursuing the M.E. degree. His research interests include image processing, programming languages, and iOS.



HIROYASU TAKAGI received the B.E. and M.E. degrees from the Nagoya Institute of Technology, in 2019 and 2021, respectively. In 2021, he joined Yamaha Corporation, Japan. His research interests include image processing and programming language.

YOSHIHIRO MAEDA (Member, IEEE) received the B.E., M.E., and Ph.D. degrees in information engineering from the Nagoya Institute of Technology, Japan, in 2013, 2015, and 2019, respectively. In 2019, he became an Assistant Professor with the Tokyo University of Science, Japan. His research interests include image-signal processing, parallel image processing, and multispectral sensing. He is a member of IEICE.

NORISHIGE FUKUSHIMA (Member, IEEE) received the B.E., M.E., and Ph.D. degrees from Nagoya University, Japan, in 2004, 2006, and 2009, respectively. He became an Assistant Professor, in 2009, and an Associate Professor, in 2015, with the Nagoya Institute of Technology, Japan. His research interests include image signal processing, parallel image processing, and compilers. He is a member of IEEE CAS, IEEE SPS, IEICE, and IPSJ.