Domain Specific Description in Halide for Randomized Image Convolution

Hiroyasu Takagi and Norishige Fukushima
Nagoya Institute of Technology, Japan

***Abstract*—We propose a domain-specific language for finite impulse response (FIR) filters using a randomized algorithm. FIR filters such as Gaussian and bilateral filters are fundamental tools for signal processing and image processing. The computational time of these convolutions increases as the convolution kernel size becomes more extensive because the computational time depends on the kernel size. Approximating the kernel by random subsampling is one of the methods to reduce the computational time. The advantage of this approximation is easily controllable for the balance between its approximation accuracy and computational time. However, it is tricky to optimize this approximation to suit the multi-core and SIMD instructions provided in recent CPUs. Halide is a domain-specific language for image processing and can powerfully accelerate image processing with a concise description. One of the remarkable points of Halide is that it can be described as a separate description of the algorithm, which is the essence of image processing, and scheduling, which optimizes the processing such as parallelization and vectorization. Using Halide, we optimize the approximation of the FIR filter by the randomized algorithm. Also, we design a new domain-specific language that outputs the optimized Halide code in a concise description.**

I. Introduction

Approaching the end of Moore's law, the computing architecture of computing units increases complexity. The complex hardware makes code optimization hard to maximize its performance. A domain-specific language (DSL) is one of the solutions. DSL for image processing is a hot topic, and various languages are proposed, such as Halide [1], Darkroom [2], PolyMage [3], [4], [5], G-API in OpenCV, and the other DSL [6], [7], [8], [9]. Halide [1] is a representative image processing DSL, and continuously developed [10], [11], [12], [13]. Halide can divide codes into two parts: an algorithm part and a scheduling part. The former specifies what to execute, and the latter determines how to execute the algorithm.

Changing only the scheduling part, we can optimize the code for the specific hardware, such as CPU (x86, ARM, MIPS, Hexagon, PowerPC, and Xeon Phi) and GPU (CUDA, OpenCL, and OpenGL). The following researches extend Halide to have MPI backend [14], DSP backend [15], and FPGA[16], [17]. For CPU backend, the Halide compiler transforms codes into LLVM IR (intermediate representation) [18] for low-level code generation and memory efficiency optimization, which improves CPU backend performance [19].

This study focuses on finite impulse response (FIR) filtering, an essential process in signal processing and is represented as a convolution. The FIR filter is the base for smoothing and edge detection processes in computational photography, computer graphics, and computer vision. The FIR filter usually exhibits $O(Sr^2)$ order, where S is the image size, and r is a filtering radius. If the kernel has separability, we can reduce the other into $O(Sr)$. For the filtering is linear time-invariant filtering, we can use fast Fourier transform (FFT) to reduce the order into $O(S \log S)$. Also, when the kernel is Gaussian, we can use short-time frequency transform to reduce the order into $O(SK)$ [20], [21], where $K < r$ is the approximation order. For more specific cases, $O(S)$ implementation is available, such as bilateral filtering [22], [23], [24], [25], [26], guided image filtering [27], [28], and high dimensional Gaussian filtering [29], [30], [31], [32], [33], [34], [35]. These cases require strict conditions for convolution kernels for order reduction.

One way to reduce the computational order for arbitrary kernel convolution is to approximate the kernel by subsampling randomly [36], [37]. This method reduces the order into $O(S \log r^2)$ and accelerates any FIR filter. The remarkable point of this approximation is easily controllable for the balance between its approximation accuracy and computational order. However, handling the randomized convolution in Halide requires a redundant description. For this case, defining a new operation is essential. For example, paper [38] adds a function of "rfactor" to handle the parallelization and the vectorization for Halide reduction operation.

In this paper, we optimize the FIR filter with a randomized algorithm using Halide. Then, we propose a new DSL description, which outputs the optimized Halide code by a concise description. We add a new description of *RandConv* for randomized convolution. Furthermore, the experimental result shows that our new DSL can describe a high-performance FIR filter with high efficiency.

II. Preliminary

A. FIR Filter

We first define FIR filter. Let two dimensional R tone image be $\boldsymbol{I} : \mathcal{S} \mapsto \mathcal{R}$, where $\mathcal{S} \subset \mathbb{R}^2$ is the spatial domain, $\mathcal{R} \subset [0, R]^c$ is the range domain, and c is the range dimension (generally, $R = 256$, and $c = 1, 3$), respectively. Let pixel position be $\boldsymbol{p} \in \mathcal{S}$, and its intensity vector be $\boldsymbol{I_p} \in \mathcal{R}$. FIR filtering with neighborhood pixels $N_{\boldsymbol{p}} \subset \mathcal{S}$ is defined as follows:

$$\bar{\boldsymbol{I}}_{\boldsymbol{p}} = \frac{1}{\eta} \sum_{\boldsymbol{q} \in N_{\boldsymbol{p}}} f_{\boldsymbol{p},\boldsymbol{q}} \boldsymbol{I_q}, \tag{1}$$

This work was supported by JSPS KAKENHI (18K19813, 21H03465).

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

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

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

    return blur_y;
}
```

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

where $\bar{\boldsymbol{I}}_{\boldsymbol{p}}$ is an output intensity value, $\boldsymbol{p}$, $\boldsymbol{q}$ is a target pixel and a reference pixel. $f_{\boldsymbol{p},\boldsymbol{q}}$ is convolutions weights, which is dependent on filtering. When the weight is only defined by spatial distance, $f_{\boldsymbol{p},\boldsymbol{q}} : \mathcal{S} \times \mathcal{S} \mapsto \mathbb{R}$. When the weight is only defined by intensity distance, $f_{\boldsymbol{p},\boldsymbol{q}} : \mathcal{R} \times \mathcal{R} \mapsto \mathbb{R}$. η is a normalization term, which is generally the sum of the weights in the kernel.

In the FIR filtering, all pixels in the kernel are referenced. Most reference pixels in natural images tend to have similar values around neighboring pixels. The similarity indicates that referencing pixels is redundant processing. Therefore, a subsampling of the kernel is considered to be possible. Kernel subsampling could be defined by substituting N for M, where M represents the set of randomly selected pixels. When $|M| << |N|$, acceleration is expected.

The weight functions are static for Linear time-invariant (LTI) filters: Gaussian, Laplacian, Gabor, and Sobel. These kernel and normalization terms can be precomputed. On the other hand, weight functions are dynamically changing for linear time-variant (LTV) filters: adaptive parameter filtering and edge-preserving filtering. LTV filter's weight cannot be precomputed because their weights vary spatially. The subsampling kernel greatly accelerates the LTV filtering since the weights should be calculated for each pixel.

B. Halide

Halide [1], [10], [11], [12] is a major DSL for image processing. The language is a pure-functional language embedded in C++. Halide can separately describe the code in algorithm parts and scheduling parts. The algorithm parts show the essence of processing and are a hardware-independent description. The scheduling parts reveal the computational order and computational method. The former example is scanning-loop order, and the latter examples are vectorization and parallelization.

Figure 1 shows the Halide code of 3×3 box filtering for CPU backend. The "*Func*" represents a pipeline stage. It is a pure function that defines what value each pixel should have. The "*Var*" is the name to be used as variables in the definition of a *Func*. The "*Buffer* src" represents an input image, and its boundaries are extended by a method in the "*BoundaryConditions*" namespace. The variables "*Var* x,y" show x and y coordinates of images and functions, and the other variables are used for the same purpose. In the algorithm parts, we horizontally average the clamped image "*clamped*", and then vertically mean the averaged image. In scheduling parts, computational scheduling is defined in each *Func* by calling various class methods, e.g., "*tile*", "*vectorize*", "*parallel*", and "*compute_at*". The *tile* method splits the image into 32×32 tiles by inner and outer variables. The *vectorize* method orders vectorized computing with single instruction, multiple data (SIMD) units: MMX, SSE, AVX, AVX-512, and NEON. This method vectorizes pixels along the xi loop. The *parallel* method shows multi-thread computing with multi-core/thread CPU, and the scheduling parallelizes along the yo loop. The *compute_at* method indicates how to memorize computed results, and we compute and memorize "*Func* blur_x" on xo loop of "*Func* blur_y" under the schedule. In the default schedule, no computation is memorized, i.e., all functions are inlined.

Halide is utilized for more complex image filtering, such as guided image filtering [16], weighted median filtering [39], and IIR filtering [40], [41], [42]. Furthermore, Halide's compiler is also applied to the field of machine learning. In work [12], Halide is extended to a differentiable programming language. Halide IR is used in TVM, a compiler for deep learning and the foundation of various deep learning frameworks. OpenCV, an image processing library, uses Halide as a backend for its deep learning module. Halide's pipelines are also run in Google's machine learning library TensorFlow, Google photo, Youtube, Pixel Visual Core in Google's smartphone Pixel 2, Adobe's Photoshop, and various other applications.

C. Description of FIR filter in Halide

1) Algorithm Part: In Halide, the convolution computing is described using "*RDom*", called the reduction domain. Halide's Func has two types of definitions: pure definition, which is a mapping from *Var* to an *Expr* (which represents the expression), and update definition, which updates a function's values. *RDom* is used in the latter of the update definition.

Figure 2 shows the Halide's algorithm code of bilateral filter for the grayscale image, which is an example of the LTV convolution. The bilateral weight is defined as follows:

$$f_{\boldsymbol{p},\boldsymbol{q}} := \exp\left(\frac{\|\boldsymbol{p}-\boldsymbol{q}\|_2^2}{-2\sigma_s^2}\right) \exp\left(\frac{\|\boldsymbol{I}(\boldsymbol{p})-\boldsymbol{I}(\boldsymbol{q})\|_2^2}{-2\sigma_r^2}\right), \quad (2)$$

where $\|\cdot\|_2$ is the L2 norm, σ_r and σ_s are standard deviations for range and spatial distributions, respectively. In the code, "*Func* conv" and "*Func* norm" represent the function of the convolution and the calculation of the normalization term, respectively. *RDom* has internal parameters "*min*" and "*extent*", and iterates computing from *min* to *min*+*extent*. "*RDom* r" is

```
Buffer<float> input = load_image();
Var x, y;

// Algorithm part
Func conv, norm, output, clamped;
RDom r(−R, 2∗R+1, −R, 2∗R+1);

clamped = BoundaryConditions::repeat_edge(input);
Expr ds = −1.f / (2.f∗sigma_s∗sigma_s);
Expr ws = exp((r.x∗r.x + r.y∗r.y)∗ds);
Expr dr = −1.f / (2.f∗sigma_c∗sigma_c);
Expr wr = exp((clamped(x+r.x, y+r.y) − clamped(x, y))∗(
      clamped(x+r.x, y+r.y) − clamped(x, y))∗dc);
Expr weight = ws∗wr;

// pure definitions
conv(x, y) = 0.f;
norm(x, y) = 0.f;
// update definitions
conv(x, y) = conv(x, y)
    + weight∗clamped(x+r.x, y+r.y);
norm(x, y) = norm(x, y) + weight;

// define output
output(x, y) = conv(x, y) / norm(x, y);
```

Fig. 2. Halide algorithm code of bilateral filter.

```
for y:
 for x:
   conv(x, y) = 0.f
for y:
  for x:
    for r.y in [−R, R]:
      for r.x in [−R, R]:
        conv(x, y) = conv(x, y) + weight∗clamped(x+r.x, y
            +r.y)
```

Fig. 3. Pseudocode of loop structure of "conv".

the reduction domain, which iterates -R to 2*R+1 in x and y dimensions. Reduction domains can be defined in multiple dimensions, and a loop over the reduction domain is generated for each dimension. Figure 3 shows a loop structure of "*Func* conv" in pseudocode.

2) Scheduling Part: Figure 4 shows Halide's scheduling code for the bilateral filter. Input image is devided into sub-images whose size is *tile_w* × *tile_h*. Then, the sub-images are processed in parallel for acceleration. Scheduling method *compute_root* specifies that all reference values will be pre-evaluated and stored. Therefore, the "clamp" that is an input image with extended edges should be specified as *compute_root*. The "conv" and "norm", which perform summation calculation, have the same loop structure. Thus, it is more efficient to merge the loops, so *compute_with* operator is used to merge the loop between the two functions. The loop structure generated by the above scheduling is shown in Fig. 5 by pseudocode. Expressions _x and _y convert inner-variables and tile index to the original x and y appropriately.

```
// Scheduling part
Var xi, yi, tile_index;

clamped.compute_root();
output.compute_root().tile(x, y, xi, yi, tile_w, tile_h)
   .fuse(x, y, tile_index).parallel(tile_index).parallel(yi);
conv.compute_at(output, xi);
norm.compute_at(output, xi).compute_with(conv, x)
   .update().compute_with(conv.update(), r.x);
```

Fig. 4. Halide scheduling code of bilateral filter

```
produce clamped:
 for y:
  for x:
   clamped(x, y) = ...
consume clamped:
 produce output:
  parallel for tile_index:
   parallel for yi:
    for xi:
     produce conv, norm:
      conv(_x, _y) = 0.f
      norm(_x, _y) = 0.f
      for r.y in [−R, R]:
       for r.x in [−R, R]:
        conv(_x, _y) = conv(_x, _y) + weight(_x, _y, r.x, r.
            y)∗clamped(_x+r.x, _y+r.y)
        norm(_x, _y) = norm(_x, _y) + weight(_x, _y, r.x,
            r.y)
     consume conv, norm
      output(_x, _y) = conv(_x, _y)/norm(_x, _y)
```

Fig. 5. Pseudocode of scheduled bilateral filter's loop structure.

III. Proposed Method

This paper proposes a description of the FIR filter with a randomly sub-sampling approximation using Halide. We also design a new DSL, which internally generates the optimized Halide code for the randomized FIR filter. Our DSL achieves an acceleration of the FIR filter with concise descriptions.

A. Kernel sub-sampling with Halide description

In Halide, kernel sub-sampling is implemented by reducing loops in *RDom*. We indicate a description, which combined a look-up table (LUT) of sub-sampled points and *RDom*.

When using sub-sampling kernels, it is vital to use the different random seeds for subsampling neighboring pixels to prevent streaking noise [36], [37]. Let the number of sampling points be $n = (X, Y) \in \mathcal{S}$, and let the number of sampling patterns be $m \in \mathbb{N}$. First, we randomly select n points, $(X_{00}, Y_{00}), (X_{01}, Y_{01}), \ldots, (X_{0n-1}, Y_{0n-1})$, according to some random selection algorithm, e.g., pure random sampling, Gaussian sampling. Then, we also randomly select n points, $(X_{10}, Y_{10}), (X_{11}, Y_{11}), (X_{12}, Y_{12}), \ldots, (X_{1n-1}, Y_{1n-1})$, in the same selection algorithm. By repeating this operation m times, we can select m pattern of n points, $(X_{m0}, Y_{m0}), (X_{m1}, Y_{m1}), (X_{m2}, Y_{m2}), \ldots, (X_{mn-1}, Y_{mn-1})$, and these are stored in a LUT including sub-sampled

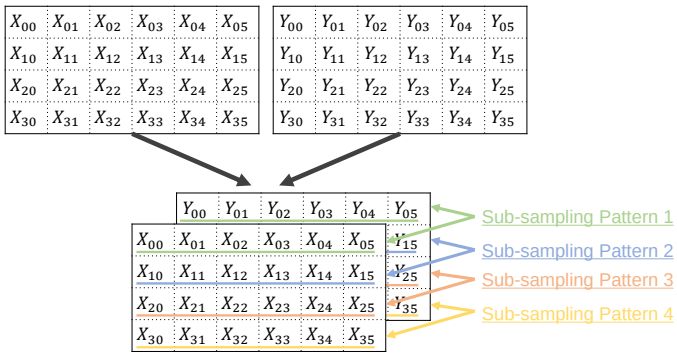


Fig. 6. LUT for kernel sub-sampling ($n = 6, m = 4$).

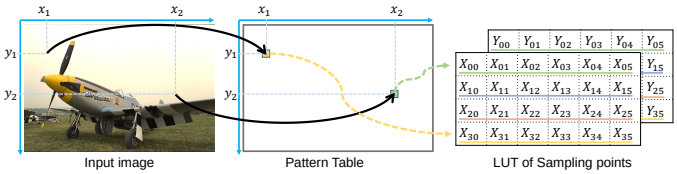


Fig. 7. How to select a sub-sampling pattern.

points. Figure 6 shows an example of the LUT created in $n = 6, m = 4$. The LUT is a three-dimensional array consisting of a set of sampled points arranged in rows. The LUT is implemented as "*Buffer<int>* LUT", where LUT(i, j, 0) indicates i-th x-coordinate selected in pattern j, and LUT(i, j, 1) indicates i-th y-coordinate selected in pattern j.

Second, we prepare a table for selecting sampling patterns. This table has the same size as the input image and is randomly selected from 0 to $m - 1$ for each value. We select a sub-sampling pattern from the value in the table that is the same coordinate as the target pixel. Figure 7 illustrates the procedure for selecting a sub-sampling pattern using the pattern table. The table is also implemented by "*Buffer<int>* pattern".

Finally, using the "LUT", "pattern", and a one-dimensional "*RDom* r_dash", which iterates from 0 to $n - 1$, the kernel sub-sampling is implemented by replace the original "*RDom* r" in Fig. 2 as follows:

line-10,12,20: r.x → LUT(r_dash, pattern(x, y), 0)
line-10,12,20: r.y → LUT(r_dash, pattern(x, y), 1)

Furthermore, for "norm" scheduling, we change the "*compute_with*" in the update definition to "*compute_with* (conv.update(), r_dash)".

B. DSL for FIR filter using randomized algorithm

We designed a new DSL to provide random kernel sub-sampling for the FIR filter in a concise description. Our DSL is built on Halide with a shorter description and easily optimizes codes. A sample code of the proposed DSL for the grayscale bilateral filter is shown in Fig. 8. For the DSL, we created new classes *RandConv* to extend the Halide functionality. The *RandConv* is a body function of the randomized convolution of FIR filtering.

The user's first step is to initialize *RandConv* by specifying the input. Then, users define weight expressions of the FIR

```
1  Buffer<float> input = load_image();
2  Var x, y;
3
4  RandConv bf("BilateralFilter");
5  bf(x, y) = input(x, y); // initialization
6
7  // define kernel
8  RDom r(−rad, 2 ∗ rad + 1, −rad, 2 ∗ rad + 1);
9  Expr ds = −1.f / (2.f ∗ sigma_s ∗ sigma_s);
10 Expr s_kernel = exp((r.x ∗ r.x + r.y ∗ r.y) ∗ ds);
11 Expr dr = −1.f / (2.f ∗ sigma_c ∗ sigma_c);
12 Expr r_kernel = exp(fast_pow(input(x + r.x, y + r.y) − input
       (x, y), 2) ∗ dc);
13
14 bf.add_kernel({ s_kernel, r_kernel } );
15 bf.bound(x, image_width).bound(y, image_height)
16   .randomize<uint8_t>(r, 30, 4, RandConv::SampleMethod
         ::Gaussian, 4.f);
17 bf.realize(output);
```

Fig. 8. *RandConv* code for bilateral filter

filter and add them to *RandConv* by *add_kernel* operator. The add_kernel operator takes vector<*Expr* > as an argument so that the kernel weight can be divided into multiple parts. Each added weight is discriminated as spatially varying or not, and different scheduling is applied to each. Also, the weight used in the filtering is the product of the divided weight parts. Finally, the size of each dimension is given to *RandConv* by *bound* operator, and randomization is applied by *randomize* operator. The arguments of randomize operator are the *RDom* to be sub-sampled, the number of sampling points, the number of sampling patterns, the value for specifying the sampling algorithm, and the sigma value for Gaussian sampling (optional). In the example shown in Fig. 8, the sampling algorithm is specified according to a Gaussian distribution with sigma equal to 4. Users can specify that the output should be cast to an arbitrary type by implementing the randomize operator as a template method.

The *RandConv* code in Fig. 8 generates a Halide function, which is almost equivalent to the kernel sub-sampled bilateral filter mentioned in Sec. III-A. The construction of the internal functions of *RandConv* is handled in the randomize operator. In the randomize operator, the kernel weights added in the *add_kernel* operator are classified into two parts: spatially varying weights or non-varying weights depending on whether the weight's expression includes the Halide's *Var* used in the input image or not. If the *Var* is included in the weight's expression, it is a spatially varying weight. For a class with the spatially non-varying weights, define the Halide's *Func* as a product of weights in the class and specify *compute_root*. If all weights are spatially non-varying, the normalization term should also be specified *compute_root* since it is possible to pre-evaluate.

Next, we substitute the *RDom* in the weight expression, as shown in Sec. III-A. Creating the LUT for the sampled points is based on the random selection algorithm specified in the argument. Moreover, substitute "r_dash" with temporary *Var* and

Table I
SPECIFICATIONS OF COMPUTER.

OS	Windows 10 Enterprise
CPU	AMD Ryzen Threadripper 3970X @ 3.7-4.5GHz (64threads)
RAM	64GB, DDR4-2666 (1333MHz)
BUILD	Microsoft Visual Studio Professional 2019

Table II
PARAMETERS SET

radius of kernel	30
number of sampling pattern	4
tiling size	256×256
sampling algorithm	Gaussian distribution

(a) input (b) output: sub-sampled (c) output: naïve

Fig. 9. Input and Outputs of bilateral filter in Halide.

this *Var* add to the argument of the weight function because Halide's *Func* does not allow using the reduction domain in its pure definition. Using the weight functions, define multiple objects of *Func* for computation of the convolution and the normalization term. The weight functions are referenced using "r_dash" in the temporary *Var* part. Finally, define a *Func* to divide the convolution and normalization term with a cast to the specified type.

IV. EXPERIMENTAL RESULTS

The input images are 512×768 grayscale images, and an example is shown in Fig. 9a. The specifications of the computer used in the experiment are shown in Table I. Also, the filtering parameters set are shown in Table II.

Figure 10 shows kernel sub-sampled bilateral filter in pure Halide implemented by us. Figure 11 shows the generated Halide *Func* code by our DSL code shown in Fig. 8. Comparing with direct implementation of Figure 10, our RandConv code 8 and naïve BF code 2 are simpler and have almost the same complexity. In addition, our DSL takes 60 percent less code than the direct implementation. Also, the loop structure generated by our DSL is shown in Fig. 12. Our DSL generates each definition of Halide *Func* for efficient randomized convolution.

Figure 13a shows a comparison of the computational time among our DSL, direct implementation of kernel sub-sampled bilateral filtering, and the full convolution, i.e., naïve implementation in Halide. Our DSL is faster than the direct implementation due to the pre-evaluate of spatially non-varying

```
1  Buffer<float> input = load_image();
2  Var x, y;
3  int rad = 30;
4  int pattern_num = 4;
5  int sampling_num = sampling_ratio*(2*rad+1)*(2*rad+1);
6  // Preparation for randomize
       implementation
7  Buffer<int> LUT(sampling_num, pattern_num, 2);
8  LUT = Gaussian_sampling(sampling_num, pattern_num,
       sigma);
9  Buffer<int> pattern(input.width(), input.height());
10 pattern = make_pattern_table(pattern.width(), pattern.height
       (), pattern_num);
11 RDom r_dash(0, sample_num, "r_dash");
12 Expr randx = LUT(r_dash, pattern(x, y), 0);
13 Expr randy = LUT(r_dash, pattern(x, y) 1);
14
15 // Algorithm part
16 Func conv, norm, output, clamped;
17 RDom r(-R, 2*R+1, -R, 2*R+1);
18 clamped = BoundaryConditions::repeat_edge(input);
19 Expr ds = -1.f / (2.f*sigma_s*sigma_s);
20 Expr ws = exp((randx*randx + randy*randy)*ds);
21 Expr dr = -1.f / (2.f*sigma_c*sigma_c);
22 Expr wr = exp((clamped(x+randx, y+randy) - clamped(x, y)
       )*(clamped(x+randx, y+randy) - clamped(x, y))*dc);
23 Expr weight = ws*wr;
24 // pure definitions
25 conv(x, y) = 0.f;
26 norm(x, y) = 0.f;
27 // update definitions
28 conv(x, y) = conv(x, y)
29     + weight*clamped(x+randx, y+randy);
30 norm(x, y) = norm(x, y) + weight;
31 // define output
32 output(x, y) = conv(x, y) / norm(x, y);
33
34 // Scheduling part
35 Var xi, yi, tile_index;
36 clamped.compute_root();
37 output.compute_root().tile(x, y, xi, yi, tile_w, tile_h)
38    .fuse(x, y, tile_index).parallel(tile_index).parallel(yi);
39 conv.compute_at(output, xi);
40 norm.compute_at(output, xi).compute_with(conv, x)
41    .update().compute_with(conv.update(), r_dash);
```

Fig. 10. Direct implementation code fot randomized bilateral filter.

weights. PSNR result for outputs of our DSL, measured as the output of naïve bilateral filter for a correct image, is shown in Fig. 13b. According to the PSNR result, our DSL achieves the same approximation accuracy as the direct implementation. Also, the computational time is about half even with a high approximation accuracy (over 50dB). Figures 9b and 9c show the output of sub-sampled bilateral filter in sampling ratio 0.1 and naïve bilateral filter, respectively. At this time, PSNR is 47.3 dB, almost imperceptible to the naked eye between the two outputs. Thus, we consider that implementation of kernel sub-sampling by Halide approximates and accelerates the processing.

These results show that our DSL implements kernel sub-sampling of the FIR filter by a concise description.

```
{
 Func BilateralFilter("BilateralFilter");
 Var x("x"), y("y");
 BilateralFilter(x, y) = uint8(((float32)conv(x, y)/(float32)
      norm(x, y)));
 }
{
 Func non_spatially("non_spatially");
 Var temp_r("temp_r"), p("p");
 non_spatially(temp_r, p) = (let t207 = LUT(temp_r, p, 0)
      in (let t208 = LUT(temp_r, p, 1) in (float32)exp_f32(
      float32(((t207*t207) + (t208*t208)))*-0.005000f)));
 }
{
 Func norm("norm");
 Var x("x"), y("y");
 norm(x, y) = 0.000000f;
 norm(x, y) = ((float32)norm(x, y) + (float32)weight(x, y,
      sampling_rdom$x));
 }
{
 Func conv("conv");
 Var x("x"), y("y");
 conv(x, y) = 0.000000f;
 conv(x, y) = (let t214 = max(min(pattern(x, y), 3), 0) in ((
      float32)conv(x, y) + ((float32)weight(x, y,
      sampling_rdom$x)*(float32)input$1(x + LUT(
      sampling_rdom$x, t214, 0), y + LUT(
      sampling_rdom$x, t214, 1)))));
}
{
 Func weight("weight");
 Var x("x"), y("y"), temp_r("temp_r");
 weight(x, y, temp_r) = (let t209 = max(min(pattern(x, y),
      3), 0) in (let t210 = ((float32)input(x + LUT(temp_r,
      t209, 0), y + LUT(temp_r, t209, 1)) - (float32)input(
      x, y)) in ((float32)non_spatially(temp_r, t209)*(
      float32)exp_f32((t210*t210)*-0.005000f))));
}
```

Fig. 11. Halide code generated by *RandConv*.

V. Conclusion

This paper discussed the description of efficient FIR filtering and proposed the kernel sub-sampling of FIR filtering in Halide. We designed a new DSL, i.e., *RandConv*, that generates the Halide code of the proposed kernel sub-sampled FIR filter by a concise description. As an experiment, we implemented a bilateral filter using our DSL. The experimental results show that our DSL accelerates a naïve implementation of a bilateral filter with sufficient approximation accuracy.

References

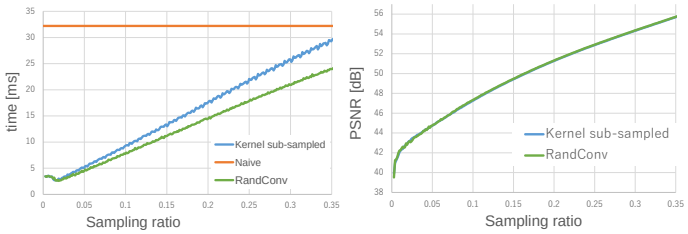
[1] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics*, 31(4):32:1–32:12, 2012-07.

[2] 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 Transactions on Graphics*, 33(4):144–1, 2014.

[3] R. T. Mullapudi, V. Vasista, and U. Bondhugula. Polymage: Automatic optimization for image processing pipelines. *ACM SIGARCH Computer Architecture News*, 43(1):429–443, 2015.


```
produce non_spatially:
  for p in [0, 3]:
    for temp_r in [0, 374]:
      kernel(...) = ...
consume non_spatially:
  produce BilateralFilter:
    parallel x.x.x_y_tile_index:
      parallel y.yi in [0, 255]:
        for x.xi in [0, 255]:
          produce conv:
            produce norm:
              conv(...) = ...
              norm(...) = ...
              for sampling_rdom in [0, 374]:
                conv(...) = ...
                norm(...) = ...
          consume sum_of_value:
            consume sum_of_total:
              BilateralFilter(...) = ...
```

Fig. 12. Loop structure pseudocode generated by *RandConv*.



(a) Computational time (b) PSNR

Fig. 13. Execution result of bilateral filter in our DSL.

[4] N. Chugh, V. Vasista, S. Purini, and U. Bondhugula. A dsl compiler for accelerating image processing pipelines on fpgas. In *Proc. International Conference on Parallel Architectures and Compilation*, 2016.

[5] A. Jangda and U. Bondhugula. An effective fusion and tile size model for polymage. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 42(3):1–27, 2020.

[6] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, and W. Eckert. Hipacc: A domain-specific language and compiler for image processing. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):210–224, 2015.

[7] M. Ravishankar, J. Holewinski, and V. Grover. Forma: A dsl for image processing applications to target gpus and multi-core cpus. In *Proc. Workshop on General Purpose Processing using GPUs*, pages 109–120, 2015.

[8] K. Selgrad, A. Lier, J. Dörntlein, O. Reiche, and M. Stamminger. A high-performance image processing dsl for heterogeneous architectures. In *Proc. European Lisp Symposium (ELS)*, 2016.

[9] 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 *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 495–504, 2018.

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

[11] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics*, 35(4):83:1–83:11, 2016.

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

[13] 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 Transactions on Graphics*.

[14] T. Denniston, S. Kamil, and S. Amarasinghe. Distributed halide. *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 51(8), February 2016.

[15] S. Vocke, H. Corporaal, R. Jordans, R. Corvino, and R. Nas. Extending halide to improve software development for imaging dsps. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):1–25, 2017.

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

[17] J. Li, Y. Chi, and J. Cong. Heterohalide: From image processing dsl to efficient fpga acceleration. In *Proc. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2020.

[18] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proc. International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2004.

[19] Y. Zhang and Y. Zhang. Making halide efficient for multicore systems. In *International Conference on Big Data Computing and Communications (BIGCOM)*, 2018.

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

[21] T. Otsuka, N. Fukushima, Y. Maeda, K. Sugimoto, , and S. Kamata. Optimization of sliding-dct based gaussian filtering for hardware accelerator. In *Proc. International Conference on Visual Communications and Image Processing (VCIP)*, 2020.

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

[23] K. N. Chaudhury. Acceleration of the shiftable o(1) algorithm for bilateral filtering and nonlocal means. *IEEE Transactions on Image Processing*, 22:1291–1300, 2013.

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

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

[26] Y. Sumiya, N. Fukushima, K. Sugimoto, and S. Kamata. Extending compressive bilateral filtering for arbitrary range kernel. In *Proc. IEEE International Conference on Image Processing (ICIP)*, 2020.

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

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

[29] A. Adams, N. Gelfand, J. Dolson, and M. Levoy. Gaussian kd-trees for fast high-dimensional filtering. *ACM Transactions on Graphics*, 28(3), 2009.

[30] A. Adams, J. Baek, and M. A. Davis. Fast high-dimensional filtering using the permutohedral lattice. *Computer Graphics Forum*, 29(2):753–762, 2010.

[31] E. S. L. Gastal and M. M. Oliveira. Domain transform for edge-aware image and video processing. *ACM Transactions on Graphics*, 30(4):69:1–69:12, 2011.

[32] E. S. L. Gastal and M. M. Oliveira. Adaptive manifolds for real-time high-dimensional filtering. *ACM Trans. on Graphics*, 31(4), 2012.

[33] P. Nair and K. N. Chaudhury. Fast high-dimensional kernel filtering. *IEEE Signal Processing Letters*, 26:377–381, 2019.

[34] T. Miyamura, N. Fukushima, M. Waqas, K. Sugimoto, and S. i. Kamata. Image tiling for clustering to improve stability of constant-time color bilateral filtering. In *IEEE International Conference on Image Processing (ICIP)*, 2020.

[35] S. Oishi and N. Fukushima. Clustering-based acceleration for high-dimensional gaussian filtering. In *Proc. Signal Processing and Multimedia Applications (SIGMAP)*, 2021.

[36] Y. Maeda, N. Fukushima, and H. Matsuo. Taxonomy of vectorization patterns of programming for fir image filters using kernel subsampling and new one. *Applied Sciences*, 8(8), 2018.

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

[38] P. Suriana, A. Adams, and S. Kamil. Parallel associative reductions in halide. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 281–291, 2017.

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

[40] G. Chaurasia, J. Ragan-Kelley, S. Paris, G. Drettakis, and F. Durand. Compiling high performance recursive filters. In *Proceedings of the 7th Conference on High-Performance Graphics*, pages 85–94, 2015.

[41] Y. Tsuji and N. Fukushima. Halide and openmp for generating high-performance recursive filters. In *Proc. International Workshop on Advanced Image Technology (IWAIT)*, 2020.

[42] H. Takagi and N. Fukushima. An efficient description with halide for iir gaussian filter. In *Proc. Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA)*, 2020.