Halide implementation of weighted median filter

Akari Ishikawa, Hiroshi Tajima, and Norishige Fukushima
Nagoya Institute of Technology, Nagoya, Japan

ABSTRACT

With the recent extension of camera applications, image filtering is essential in image processing. Weighted median filtering is one of the image denoising method. The weighted median filter can be more useful for removing noise and blurring correction; however, its computational cost is high. Halide is a domain-specific language for image processing. By using Halide, we can easily optimize the code of image processing. In this study, we present weighted median filter with Halide code. Experimental results show that we can easily write the weighted median filter code.

Keywords: Halide, domain-specific language, median filter, weighted median filter

1. INTRODUCTION

With the recent extension of camera applications, image filtering is essential in image processing, such as image denoising and detail enhancement. Edge-preserving filtering is important for such applications, such as median filtering[1] and its variants[2,3], bilateral filtering[4] and its acceleration methods[4-9], non-local means filtering[10], guided image filtering[11.12], domain transform filtering[13], and recursive bilateral filtering[14].

The most of these filters are categorized into finite impulse response (FIR) filtering and infinite impulse response (IIR) filtering. These filters have some efficient ways to write efficient codes of FIR[15-17] and IIR[18] filtering. However, statistic filtering, such as median filtering, has complex computing order; thus, optimizing the statistic filtering is hard. Median filtering[1] is one of the statistic filtering for image denoising. The median filter replaces a pixel with the median value of the pixel's kernel. Also, weighted median filtering[2, 3] is an extension of the median filter. The filter had weights to each pixel in the kernel for statistic computing values. The difference between the two filters is the removal capability of the impulsive noise. The weighted median filter can be more useful for removing noise and blurring correction. The drawback of the weighted median filter is its computational complexity.

Approaching the end of Moore's law, CPU microarchitectures become more complex year by year; therefore, it is difficult to write programs suited to each execution environment. Halide is one of a solution to this problem. The Halide[20-23] is a domain-specific language (DSL) for image processing and a pure functional language but embedded in C++. Halide code is modularized as algorithm parts and scheduling parts. By using Halide, we can easily optimize the code of image processing by the modularization. In addition, the language is extended for recursive filtering[24] and FPGA[25].

In this paper, we focus on the statistic filtering of median filtering, and present algorithm parts of the median and weighted median filter written in Halide. The implemented code is easy for us to write the weighted median filter code. Also, only by adding the Halide scheduling code on this weighted median filter, we can efficiently parallelize the filter by slightly modifying the code.

2. HALIDE

The Halide code is modularized as algorithm parts and scheduling parts. This makes the optimization of the Halide code flexible. The algorithm parts show the image processing algorithm, and the scheduling parts reveal the computational order and computational method, e.g., loop-unrolling, loop-interchanging, loop-fission/fusion, loo-splitting, loop collapsing, tiling, vectorization, and parallelization.

Figure 1 shows the Halide code of 3×3 box filtering. *Func* indicates equations and *Var* shows variables. "*Func f*" represents an input image, and "*Var* x, y" show *x* and *y* coordinates of images, respectively. In the algorithm parts, we

horizontally average the input image *f*, and then vertically mean the averaged image. In the scheduling parts, computational scheduling is defined for each equation of *Func* by calling various class methods, e.g., *tile*, *vectorize*, *parallel*, and *compute_at*. *tile* points image tiling, and the scheduling splits the image into 256×32 tiles. *vectorize* orders vectorized computing with SIMD units, e.g., SSE and AVX, and this vectorizes pixels along the *x* loop. *parallel* shows multi-thread computing with multi-core/thread CPU, and the scheduling parallelize along the *y* loop. *compute_at* indicates how to memorize computed results, and we compute and memorize "*Func* blur_x" on *x*, *y* in ranged computation of "*Func* blur_y" under the schedule. In the default schedule, no computation is memorized, i.e., all functions are re-computed.

```
Func blur 3x3(Func f )
{
   Func blur_x , blur_y;
   Var x , y , xi , yi;

   // algorithm part
   blur x (x , y) = (f(x−1, y) + f(x, y) + f(x+1, y )) / 3;
   blur_y (x, y) = (blur_x(x, y−1) + blur_x(x, y) + blur_x(x, y+1)) / 3;

   // scheduling part
   blur_y.tile (x, y, xi, yi ,256, 32).vectorize(xi, 8).parallel(y);
   blur_x.compute_at(blur_y, x).vectorize(x, 8);

   return blur_y;
}
```

Figure 1. Halide code of 3×3 box filtering.

3. IMPLEMENTATION

Figure 2 shows the algorithm parts of the weighted median filter with bilateral filter's weight[3] written in Halide. Note that we do not write the scheduling part in Fig. 2. "*Var* x, y" show *x* and *y* coordinates of images, "*Var* l" shows a number of rows of a cumulative sum vector, and "*Var* xr, yr" show *x* and *y* coordinates of kernels for computation of the bilateral weight. *RDom*(*min*, *range*) shows variables for putting update passes inside loops in the range of *min* to *min* + *range* − 1. *select* is similar to the ternary operator in C language. If the first argument is true, then return the second, else return the third. *d_gamma* and *c_gamma* are smoothing parameters, and *r* is a kernels radius for computation of the bilateral weight. An input image "*Func* I" is a grayscale image.

At first, we compute a bilateral weight in *maskVals* under the declaration of variables. Next, we generate histograms with a horizontal axis as pixel data and vertical axis as a rate. And compute the total sum of histogram in *hist_sum* and the cumulative sum of each histogram in *hist_cumsum*. Median values equal to pixel values *R2* of cumulative sum over a half of the total sum at first. Therefore, the weighted median filter is generated with *R2* as median values. Finally, "*Func* median" is computed in the range of *width* × *height* by implementing "median.realize(*width*, *height*)". The function call Halide compiler and run the part of Halide code. When we determine the specific scheduling of this filter, we will insert the scheduling code before the function of "realize". The effective scheduling of this filter will be shown in the next section of experimental results.

```
Func bilateralWeight(Func& img)
{
// declaration of variables;
   Func d_diff,c_diff,weight;
   Var x,y,xr,yr;
   float d_norm = -2 * d_gamma * d_gamma;
   float c_norm = -2 * c_gamma * c_gamma;
// algorithm part
   d_diff(xr, yr) = exp((xr * xr + yr * yr) / d_norm);
   c_diff(xr, yr, x, y) = exp(pow(abs(img(x, y) - img(x+xr, y+yr)), 2) / c_norm);
   weight(xr, yr, x, y) = d_diff(xr, yr) * c_diff(xr, yr, x, y);
// scheduling part

   return weight;
}

Func weightedMedianFilter(Func& I)
{
// declaration of variables;
   Func maskVals, hist, hist_sum, hist_cumsum, median;
   Var x, y, l;
   RDom R(-r, 2*r+1, -r, 2*r+1), R1(1, 255), R2(0, 255);
// algorithm part
// computing bilateral weight
   maskVals(xr, yr, x, y) = bilateralWeight(I)(xr, yr, x, y);
// updating histogram
   hist(l, x, y) = 0;
   hist(I(x+R.x, y+R.y), x, y) += maskVals(R.x, R.y, x, y);
   hist_sum(x, y) = sum(maskVals(R.x, R.y, x, y));
   hist_cumsum(l, x, y) = hist(0, x, y);
   hist_cumsum(R1, x, y) = hist(R1, x, y) + hist_cumsum(R1-1, x, y);
// getting median value
   median(x, y) = -1;
   median(x, y) = select(median(x, y) < 0, select(hist_cumsum(R2, x, y) > hist_sum(x, y) / 2, R2, -1), median(x, y));
// scheduling part
// compiling or running Halide code.  Scheduling part are inserted here
   median.realize(wid, hei);
   return median;
}
```

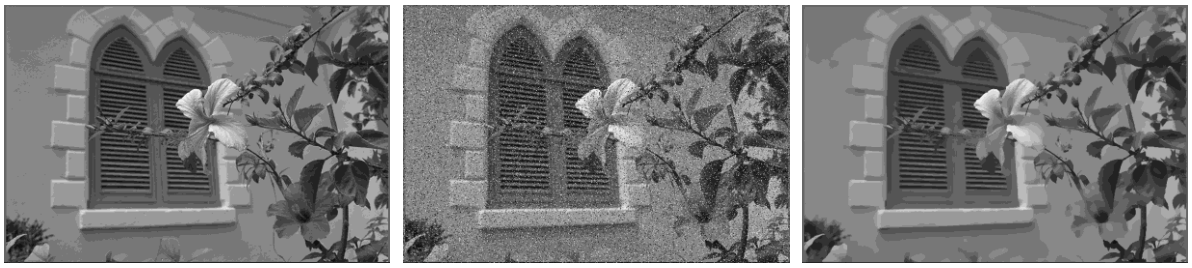
Figure 2. Implementation of weighted median filter in Halide.

4. EXPERIMENTAL RESULTS

In our experiment, we show the effectiveness of the Halide for weighted median filtering. In the weighted median filter, the input image was 768×512 grayscale image. The parameters of the filter were r=2, *d_gamma*=100 and *c_gamma*=150. CPU was Intel Core i7-7800 3.50 GHz compiled with Visual Studio 2017.

Figure 3 is the result of the weighted median filter against the salt and pepper noise image. We needed an hour to obtain the result of the weighted median filter without scheduling parts, which is shown in Fig. 2. Accordingly, for fast computation, we added scheduling parts. In Fig. 4, "*compute_root*" indicates that associated functions are computed all

once ahead of time. "*reorder"* swaps the order of variables to have the given nesting order. "*update*" is used to optimize updated *Func*. In case of adding Figure 4, the computational time of the code was improved to 31ms. Besides, the code length of the algorithm part was 45 lines, and the code length of the scheduling part was 10 lines.



(a) Original image. (b) Add noise (c) Weighted median

Figure 3. Result of weighted median filtering: (a) The input image, (b) Corrupted (a) by spike noise, (c) The weighted median filtering result of (b).

```
Func bilateralWeight(Func& img)
{
    // algorithm part
    . . .
    //scheduling part
    distance_diff.compute_root().parallel(yr).vectorize(xr, 16);
}
Func weightedMedianFilter(Func& I)
{
    // algorithm part
    . . .
    //scheduling part
    hist.compute_at(medianVals, y).vectorize(x, 16).update(0).reorder(R.x, x, R.y, y).vectorize(x, 16);
    hist_sum.compute_at(medianVals, y).vectorize(x, 16);
    hist_cumsum.compute_at(medianVals, y).vectorize(I, 16).update(0).vectorize(x, 16);
    medianVals.vectorize(x, 16).parallel(y).update(0).parallel(y).vectorize(x, 16);
}
```

Figure 4. Scheduling parts added to the code of Figure 2.

5. CONCLUSION

In this paper, we presented the weighted median filter with Halide code. By using Halide, we were able to easily write the code. Only by adding Halide scheduling in the code, we improved computational time. In the effective scheduling needed 10 lines, computational time of the code changed to 31 ms, we were able to efficiently parallelize the filter by slightly modifying the code.

ACKNOWLEDGEMENT

This work was supported by KAKENHI JP17H01764, JP18K19813.

REFERENCES

[1] Tukey, J. W., "Non-linear (non-superposable) methods for smoothing data," Congr. Rec. 1974 EASCON 673 (1974).

[2] Lin, Y., Yang, R., Gabbouj, M., and Neuvo, Y., "Weighted median filters: a tutorial," IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing, 43, 3, 157-192 (1996).

[3] Nguyen, V. A., Min, D., and Do, M. N., "Efficient techniques for depth video compression using weighted mode filtering," IEEE Transactions on Circuits and Systems for Video Technology, 23, 2, 189-202 (2012).

[4] Tomasi, C. and Manduchi, R., "Bilateral filtering for gray and color images," International Conference on Computer Vision (ICCV) (1998).

[5] Porikli, F., "Constant time o(1) bilateral filtering," IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2008).

[6] Yang, Q., Tan, K. H., and Ahuja, N., "Real-time o(1) bilateral filtering," IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2009).

[7] Chaudhury, K. N., Sage, D., and Unser, M., "Fast o(1) bilateral filtering using trigonometric range kernels," IEEE Transactions on Image Processing, 20, 12, 3376–3382 (2011).

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

[9] Sugimoto, K., Fukushima, N., and Kamata, S., "200 FPS constant-time bilateral filter using SVD and tiling strategy," IEEE International on Image Processing (ICIP) (2019).

[10] Buades, A., Coll, B., and Morel, J.-M., "A non-local algorithm for image denoising," IEEE Computer Vision and Pattern Recognition (CVPR) (2005).

[11] He, K., Sun, J., and Tang, X., "Guided image filtering," European Conference on Computer Vision (ECCV) (2010).

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

[13] Gastal, E. S. L. and Oliveira, M. M., "Domain transform for edge-aware image and video processing," ACM Transactions on Graphics, 30, 4, (2011).

[14] Yang, Q., "Recursive bilateral filtering," European Conference on Computer Vision (ECCV) (2012).

[15] Maeda, Y., Fukushima, N., and Matsuo, H., "Effective implementation of edge-preserving filtering on CPU microarchitectures," Applied Sciences, 8, 10, 1985 (2018).

[16] Maeda, Y., Fukushima, N., and Matsuo, H., "Taxonomy of vectorization patterns of programming for FIR image filters using kernel subsampling and new one," Applied Sciences, 8, 8, 1235 (2018).

[17] Fukushima, N., Tsubokawa, T., and Maeda, Y., "Vector addressing for non-sequential sampling in FIR image filtering," IEEE International on Image Processing (ICIP) (2019).

[18] Fukushima, N., Maeda, Y., Kawasaki, Y., Nakamura, M., Tsumura, T., Sugimoto, K., and Kamata, S., "Efficient computational scheduling of box and Gaussian FIR filtering for CPU microarchitecture," Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA) (2018).

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

[20] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S., "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2013).

[21] Mullapudi, R. T., Adams, A., Sharlet, D., Ragan-Kelley, J., and Fatahalian, K., "Automatically scheduling Halide image processing pipelines," ACM Transactions on Graphics, 35, 4, 83 (2016).

[22] Li, T.-M., Gharbi, M., Adams, A., Durand, F., and Ragan-Kelley, J., "Differentiable programming for image processing and deep learning in Halide," ACM Transactions on Graphics, 37, 4 (2018).

[23] Chaurasia, G., Ragan-Kelly, J., Paris, S., Drettakis, G., and Durand, F., "Compiling high performance recursive filters," High-Performance Graphics (HPG) (2015)

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