Performance Evaluation of Halide Auto-Scheduler with Directional Cubic Convolution Interpolation

Haruki Nogami[a], Sou Oishi[a], Tomohiro Sasaki[a], Yoshihiro Maeda[b], and Norishige Fukushima[a]

[a]Nagoya Institute of Technology, Gokiso-Cho, Showa-Ku, Nagoya, Aichi, 466–8555, Japan.
[b]Tokyo University of Science, 6-3-1 Niijuku, Katsushika-ku, Tokyo 125-8585, Japan.

ABSTRACT

Finding the optimal implementation of calculations is one of the most critical challenges in image processing programming. Halide is a domain-specific language for high-performance image processing. Its auto-scheduler is a helpful tool for solving this problem; however, its scheduling is not a panacea for complex flows. In this paper, we evaluate the performance of the auto-scheduler by comparing it to hand-manually implemented C++ codes. The algorithm used for comparison is Directional cubic convolution interpolation (DCCI), whose computation schedule is challenging to optimize. We evaluate three auto-schedulers: Adams et al.'s, Li et al.'s, and Mullapudi et al.s. Experimental results show that the performance of the schedule generated by Adams' method is comparable to that of the hand-implemented C++ code.

Keywords: Halide, Auto-Scheduler, Directional Cubic Convolution Interpolation

1. INTRODUCTION

High-performance image processing should effectively utilize computational resources. However, the complexity of computer architecture is increasing every year. It is difficult to write optimal programs because each architecture has different clock frequencies, the number of cores, cache sizes, and available vector arithmetic instructions. Therefore, writing high-speed image processing programs requires advanced knowledge and experience with image processing and hardware. There are various tasks in image processing to optimize these code, such as filtering,[1–3] wavelet,[4] and K-means;[5] however, these codes tends to be complex.

Halide[6,7] is a domain-specific language for image processing and is one of the solutions to this problem. Halide's codes have two parts: algorithms and schedules. The algorithm parts determine what to compute. The schedule parts determine how to compute. Optimizing the schedule improves execution speed. Auto-scheduler[8–10] is an automatic scheduling algorithm. There are three methods implemented in Halide standard library. However, the auto-scheduler only sometimes generates the optimal schedule. Therefore, it is helpful for image processing programmers to know to what extent these methods can speed up image processing.

Directional cubic convolution interpolation (DCCI)[11] is a type of algorithm whose computation schedule is challenging to optimize. It is an upsampling algorithm that interpolates pixels in two steps. The first step interpolates some pixels by referring to the original image. The second step interpolates the remaining pixels by referring to the pixels interpolated in the first step and the original image. The dependency of interpolated pixels between each step makes optimizing the DCCI computation schedule difficult.

This paper describes various codes and compares each implementation's speeds and code lengths. The prepared codes are C++, C++ with SIMD optimization, and Halide using the three scheduling methods with auto-scheduler. Through this comparison, we evaluate how well the Halide auto-scheduler contributes to the optimization of DCCI.

Further author information: (Send correspondence to N. Fukushima)
N.Fukushima: Web: https://fukushima.web.nitech.ac.jp/en/. This work was supported by JSPS KAKENHI (21H03465). Y.Maeda: Web: https://sites.google.com/view/yoshihiromaeda/. This work was supported by JSPS KAKENHI (21K17768).

```
Func blur_3x3(Func input) {
  Func blur_x, blur_y;
  Var x, y, xi, yi;

  // The algorithm part
  blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
  blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1))/3;

  // The schedule 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

2. RELATED WORK

2.1 Halide

Halide[6,7] is a domain-specific language for high-performance image processing for multi-platform, such as CPU, GPU[6,7] and FPGA.[12] The language is a pure functional language and is embedded in C++. A Halide program consists of the following two parts: the algorithm part and the schedule part. The algorithm part is the essence of image processing and represents what to compute. This part defines the formula of the output pixel. The schedule part is the specific computation order of image processing and represents how to compute. This part defines a balance of trade-offs between parallelism, memory locality, and redundant computation. By separating the algorithm from the schedule, the programmers can try to speed up image processing by rewriting only the scheduling part. Halide is adapted for various image processings, such as finite impulse convolution,[13] infinite impulse convolution,[14] recursive convolution,[15] and statical convolution.[16]

Figure 1 shows an example of Halide's code, which is a 3×3 box filter. *Func* is Halide's function, representing one stage of the Halide pipeline. *Var* is Halide's dimension variable used when defining *Func*. The algorithm part defines a separable box filter. *input* represents the input image. *blur_x* calculates the average of three horizontal pixels of *input*. *blur_y* calculates the average of three vertical pixels of *blur_x*. In the schedule part, *blue_y* is tiled with tile size 256×32, vectorized by 8 pixels in the innermost loop and parallelized outermost loop. *blur_x* is scheduled to be computed in *blur_y*'s every tile and vectorized by 8 pixels. Thus, Halide allows the algorithm and the schedule to be defined separately.

Auto-scheduler[8–10] is an automatic scheduling algorithm. It generates a schedule of image processing programs based on several parameters. Examples of parameters are the expected size of input and output images, the maximum number of parallel operations, the capacity of the last-level cache, and the cost of a cache miss in the last-level cache. Parameter values and the kind of auto-scheduler method can be switched easily by command line arguments at compile time. Therefore, there is no need to rewrite the Halide code when changing parameters and auto-scheduler methods. There are three auto-scheduler methods built in Halide's standard library: Mullapudi et al.'s works,[10] Li et al.'s works,[9] and Adams et al.'s works.[8]

2.1.1 Auto-scheduler of Mullapudi's method.

Mullapudi's method[10] is an automatic scheduling method using the greedy algorithm. Halide represents image processing as a directed acyclic graph (DAG) of computations. Mullapudi's method groups the nodes of this DAG and performs tiling for each group. Large group size increases redundant computations due to tiling. Smaller groups increase memory usage to store the calculation results. Mullapudi's method balances this trade-off using the greedy algorithm.

2.1.2 Auto-scheduler of Li's method

Li's method[9] is specialized to handle many reduction operations, such as gradient computation in deep learning. Li's method schedules function as follows.

1. A function used by multiple functions is computed once ahead of time.
2. A function used by only one function expands inline.

2.1.3 Auto-scheduler of Adams' method

The variety of possible schedules for image processing algorithms is enormous. For this reason, Mullapudi's method places various constraints on search schedules. For example, Halide allows tile sizes to be specified for each stage of Halide's pipeline, while Mullapudi's method only allows the same tile size for a group. These constraints keep the search time low. Adams' method[8] relaxes this constraint. Therefore, selecting a schedule efficiently from various possible schedules is necessary. Adams' method solves this problem using beam search and a hybrid cost model of symbolic analysis and machine learning.

2.2 Directional Cubic Convolution Interpolation (DCCI)

DCCI[11] is an upsample algorithm that considers edge directions. Figure 2 shows the interpolation flow of DCCI.

At first, DCCI copies the input image $I_{i,j}$ to the output image $I'_{2i,2j}$ where i, j are the coordinates of the input image. DCCI interpolates pixels in two steps. The first step interpolates $I'(2i+1, 2j+1)$. The second step interpolates $I'(2i, 2j+1), I'(2i+1, 2j)$. In both the first and second steps, DCCI's interpolation process consists of gradient and interpolation calculations.

The gradient calculations are

$$G_1(i', j') = \begin{cases} \sum_{m=3,\pm1} \sum_{n=3,\pm1} |I'_{i'+m,j'-n} - I'_{i'+m-2,j'-n+2}| & \text{if step 1} \\ \sum_{m=\pm1} \sum_{n=0,2} |I'_{i'+m,j'-n} - I'_{i'+m,j'-n+2}| + \sum_{m=0,\pm2} |I'_{i'+m,j'-1} - I'_{i'+m,j'+1}| & \text{if step 2} \end{cases} \tag{1}$$

$$G_2(i', j') = \begin{cases} \sum_{m=3\pm1} \sum_{n=3,\pm1} |I'_{i'+m,j'+n} - I'_{i'+m-2,j'+n-2}| & \text{if step 1} \\ \sum_{m=0,2} \sum_{n=\pm1} |I'_{i'-m,j'+n} - I'_{i'-m+2,j'+n}| + \sum_{n=0,\pm2} |I'_{i'-1,j'+n} - I'_{i'+1,j'+n}| & \text{if step 2} \end{cases} \tag{2}$$

where i', j' are the coordinates of the output image.

In the first step, G_1 is a 45-degree gradient, and G_2 is a 135-degree gradient. In the second step, G_1 is the vertical gradient, and G_2 is the horizontal gradient.

The interpolation calculations are

$$I'_{i',j'} = \begin{cases} p_2 & \text{if } (1+G_1)/(1+G_2) > T \\ p_1 & \text{if } (1+G_2)/(1+G_1) > T \\ (w_1 p_1 + w_2 p_2)/(w_1 + w_2) & \text{otherwise} \end{cases} \tag{3}$$

where T is the threshold for edge determination. The parameter of p_1 and p_2 are

$$p_1 = \begin{cases} \text{45-degree directional 1-D cubic convolution interpolation nearby } I'_{i',j'} & \text{if step 1} \\ \text{vertical directional 1-D cubic convolution interpolation nearby } I'_{i',j'} & \text{if step 2} \end{cases} \tag{4}$$

$$p_2 = \begin{cases} \text{135-degree directional 1-D cubic convolution interpolation nearby } I'_{i',j'} & \text{if step 1} \\ \text{horizontal directional 1-D cubic convolution interpolation nearby } I'_{i',j'} & \text{if step 2} \end{cases} \tag{5}$$

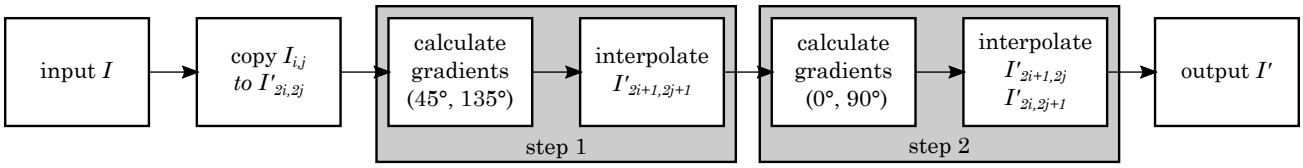


Figure 2: Algorithm flow of DCCI.

$$w_1 = \frac{1}{1 + G_1^k} \tag{6}$$

$$w_2 = \frac{1}{1 + G_2^k} \tag{7}$$

where k is a parameter used for weight adjustment.

DCCI implemented as per the algorithm has a problem regarding the computation schedule. This method places interpolated pixels sparsely on the output image at the first step. Therefore, the efficiency of vector operations is low. In addition, scanning the entire image multiple times reduces locality. As a result, cache misses increase.

3. EXPERIMENTAL CODES

This paper compares the following types of codes: Naïve C++, vectorized C++, and Halide. The naïve C++ implementations just use C++ with/without parallelization by OpenMP (C++-Naïve/C++-Naïve (parallelized)). The vectorized C++ implementation utilizes AVX2 for SIMD vectorization (C++-SIMD). We used Visual Studio 2019 to compile these C++ codes. In addition, Halide's auto-scheduler uses three methods: Adams' one,[8] Li's one,[9] and Mullapudi's one.[10] So, we compared the implementations output by the auto-schedulers of Adams' (Halide-Adams), Li's (Halide-Li), and Mullapudi's (Halide-Mullapudi). We compiled them using Halide13.04 and scheduled them individually according to the input image size.

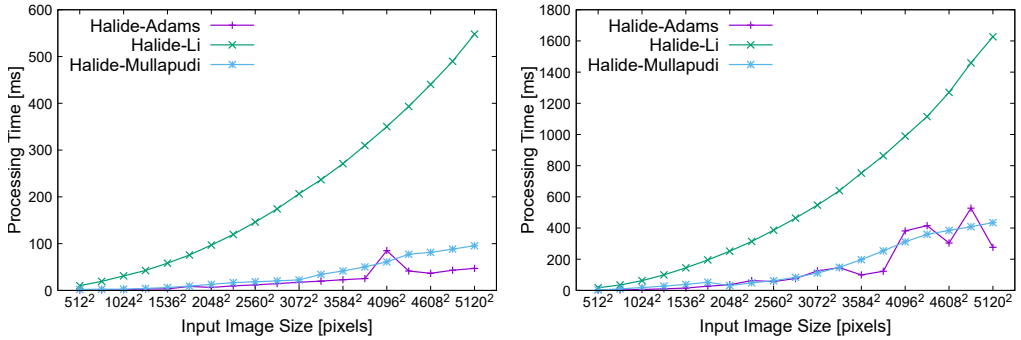
Table 1 shows the number of lines of these codes. We count the number of lines of code written by ourselves, excluding the C++ and the Halide standard library. The code layout is designed for easy viewing. Thus, the code length should be used only as a reference. The naïve C++ implementation code is the shortest. The vectorized C++ implementation code is the longest. Compared to the vectorized C++, Halide implementations' code is 0.58 times shorter for gray images and 0.31 times shorter for color images. Also, the Halide implementation code is common to each auto-schedulers.

4. EXPERIMENTAL RESULTS

Figure 3 shows the average processing time for 1000 times upsampling of various-sized images. We increased the size of the experimental images from 512×512 to 5120×5120 in increments of 256 per side. We measured them on an AMD Ryzen9 5950X 3.4GHz 16core 32threads CPU. The results show that Halide-Li is significantly slower

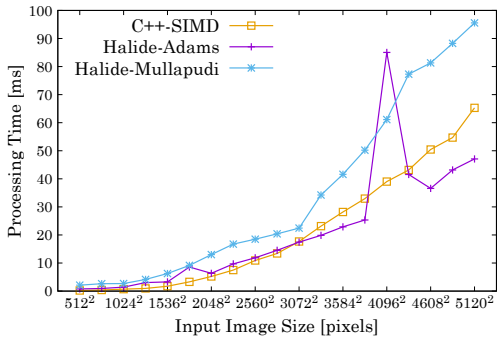
Table 1: Code Length [line].

	gray	color
C++-Naïve	90	135
C++-Naïve (parallelized)	92	137
C++-SIMD	253	488
Halide (Common to Halide-Adams, Halide-Li, and Halide-Mullapudi)	147	156

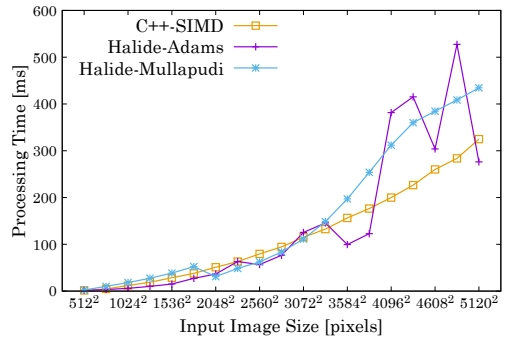


(a) Halide scledules. (Gray)

(b) Halide scledules. (Color)



(c) Halide and C++ codes. (Gray)



(d) Halide and C++ codes. (Color)

Figure 3: Processing time for each input image size in the case of a 2× upsample. As a reference, C++-Naïve 2286.3 ms, C++-Naïve (parallelized) 395.3 ms for 5120 × 5120 gray image 2× upsample. C++-Naïve 67373.7 ms, C++-Naïve (parallelized) 6249.0 ms for 5120 × 5120 color image 2× upsample.

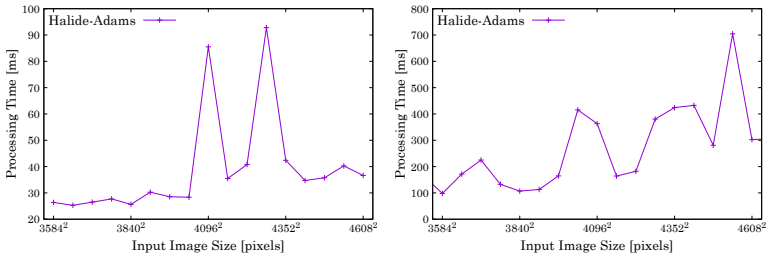
than the other two methods for gray and color images. Halide-Adams is generally faster than Halide-Mullapudi, although Halide-Mullapudi is faster than Halide-Adams for some image sizes.

Halide-Adams, the fastest implementation in auto-schedulers, was also faster than the implementation with manual optimization in SIMD. However, the Halide-Adams performed significantly slower for specific image sizes, such as 4096×4096 gray images and larger sizes of color images than 4096×4096. Halide-Li and Halide-Mullapudi did not show this tendency.

We conducted additional experiments to verify the instability of Halide-Adams. We used image sizes ranging from 3584×3584 to 4608×4608 with 64 per side increments, which are smaller increments than the first experiments. Schedules were generated individually by setting each image size as a parameter of the auto-scheduler. Figure 4 shows the average processing time for 1000 times upsampling. The results showed that the performance of the generated schedules by Adams' method varied significantly as the input image size varied. It indicates that when Adams' method outputs a slow schedule, a slight change in parameters may improve the performance of the output schedule.

5. CONCLUSION

We evaluated the performance of the halide auto-scheduler using DCCI. The results showed that Adams' method performed best among the auto-schedulers in DCCI. However, the performance of Adams' method may vary depending on parameters such as image size. In such cases, it is possible to improve the performance by slightly changing the parameters. In addition, we showed that the Halide auto-scheduler could achieve performance comparable to vectorized C++ implementation with significantly less code.



(a) Gray (b) Color

Figure 4: Processing time of Halide-Adams for each input image size in the case of a 2× upsample.

REFERENCES

[1] Kondo, T., Maeda, Y., and Fukushima, N., "Accelerating finite impulse response filtering using tensor cores," in [*Proc. Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*], (2021).

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

[3] 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) (2018).

[4] Sumiya, Y., Kamei, H., Ishikawa, K., Sumiya, Y., and Fukushima, N., "Vectorized computing for edge-avoiding wavelet," in [*International Workshop on Advanced Image Technology (IWAIT),*], **12177**, 23 – 28, International Society for Optics and Photonics, SPIE (2022).

[5] Otsuka, T. and Fukushima, N., "Vectorized implementation of k-means," in [*Proc. International Workshop on Advanced Image Technology (IWAIT)*], (2021).

[6] 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), 32:1–32:12 (2012-07).

[7] 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," in [*Proc. ACM Programming Language Design and Implementation (PLDI)*], (2013).

[8] Adams, A., Ma, K., Anderson, L., Baghdadi, R., Li, T., Gharbi, M., Steiner, B., Johnson, S., K.Fatahalian, Durand, F., and Ragan-Kelley, J., "Learning to optimize halide with tree search and random programs," *ACM Transactions on Graphics* **38**(4), 121:1–121:12 (2019).

[9] Li, T., 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), 139:1–139:13 (2018).

[10] 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), 83:1–83:11 (2016).

[11] Zhou, D., Shen, X., and Dong, W., "Image zooming using directional cubic convolution interpolation," *IET Image Processing* **6**(6), 627–634 (2012).

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

[13] Takagi, H. and Fukushima, N., "Domain specific description for randomized image convolution," in [*Proc. Asia-Pacific Signal and Information Processing Association Annual Summit and Conference*], (2021).

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

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

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