Hisui: an Image and Video Processing Framework with Auto-optimizer

Ippei OKUDA*, Masahiro TAKAOKA* and Tomoaki TSUMURA*
* Nagoya Institute of Technology, Japan
E-mail: camp@matlab.nitech.ac.jp

***Abstract*—In order to develop a high-performance image and video processing program, it is necessary to tune the program to a target platform. However, this task not only requires a high level of knowledge of computer architecture and parallel processing, but also requires an enormous amount of development time. It makes the cost of program development a significant problem. To solve this problem, we are developing Hisui that aims to achieve both high productivity and high performance in the development of image and video processing programs. Hisui consists of a highly abstract Hisui language and Hisui compiler that compiles programs written using Hisui language. In this study, we design an automatic optimization function for Hisui and implement it on Hisui compiler. This function can automatically apply multi-threading, vectorization, and cache blocking. In order to design this function, we first investigated these optimization methods and identified the appropriate optimization application criteria and methods. Then, based on the results of the survey, we design an automatic optimization function and implement it on Hisui compiler. As a result of evaluating the usefulness of the improved image and video processing framework, we confirmed that the programs optimized by Hisui compiler are up to 3.0 times faster than those optimized by a combination of Halide and its Auto-scheduler.**

I. Introduction

Image and video processing technology has developed and is applied in many fields. As a result, opportunities to develop image and video processing programs are increasing. In order to develop high-performance image and video processing programs, it is necessary to tune the programs to the platform.

However, tuning image and video processing programs requires considerable development costs. It is necessary to require a high level of knowledge about computer architecture and parallel processing. In addition, in order to find the optimal parameters, such as the number of threads for parallel processing, it is necessary to implement programs with various parameters and verify their performance. This requires advanced programming techniques and an enormous amount of development time, placing a heavy burden on the programmers. In addition, since the optimal tuning parameters differ from platform to platform, the portability of the tuned program is reduced.

To overcome this problem, We have proposed an image and video processing environment, Hisui [1], which aims to maximize the performance of the platform with a concise description. This environment consists of a highly abstract Hisui language and Hisui compiler that performs advanced optimization on programs written in the language.

In this paper, we introduce the outline of Hisui language and the design of dedicated auto-optimizer for Hisui. We investigated appropriate parameters for some types of optimization, and implemented Hisui compiler that automatically performs high-level optimization on programs based on the parameter investigation. Then, we evaluate the performance of programs optimized by Hisui compiler and demonstrate the usefulness of Hisui.

II. Related Work

Various frameworks specialized for developing image and video processing programs [2], [3], [4], [5], [6], [7] have been proposed. In this section, we provide an overview of Halide which is one of the best frameworks in terms of both descriptiveness and performance.

One of the most popular image processing environments today is Halide [8]. Halide is a domain-specific programming language (DSL) designed to write image and array processing codes in a concise manner, and is implemented as a functional language to be embedded in C++. Halide supports various platforms such as multi-core CPUs, GPUs, and mobile processors, etc. The most important feature of Halide is that the "algorithm" of image processing and the "schedule" of computation order and data alignment can be written separately. This makes it easy to write programs with various schedules without changing the algorithm. For this reason, programs written in Halide are relatively more readable than those written in general programming languages. Halide also provides built-in functions for scheduling that allow programmers to use various schedules in a concise manner. As a result, the time required to find the optimal schedule can be greatly reduced with Halide.

However, even with Halide, the task of specifying an optimal schedule is not easy. In order to find the optimal schedule, it is necessary to understand the computer architecture and various optimizations, select the appropriate function among the many built-in functions provided by Halide, and specify the optimal parameters. Since this is not an easy task for programmers who are not familiar with image processing or computer architectures, Halide provides an Auto-scheduler that automatically applies appropriate schedules to the algorithms written by the programmer. [9]. Using this method, the programmer only needs to write the algorithm, and the Auto-scheduler estimates the appropriate schedules and optimizes the program, thus generating a program with a certain level of

```
for(y = 0; y < 480; y++) {
    for(x = 0; x < 640; x++) {
        ave = ( img[x][y].R + img[x][y].G + img[x][y].B ) / 3;
        img[x][y].R = ave;
        img[x][y].G = ave;
        img[x][y].B = ave;
    }
}
```

(a) code in C++.

```
(pixel)p1@(image)img1{
    ave=(p1.R + p1.G + p1.B)/3;
    p1.{R, G, B}={ave, ave, ave};
}
```

(b) code in Hisui language.

Fig. 1. An image conversion code.

performance. However, the Auto-scheduler is currently unable to provide an optimal schedule for all algorithms [10]. Also, since Halide does not provide the ability to bind threads to cores, the cache held by each core cannot be used effectively, and performance may not improve much even when parallel processing is performed by multiple threads.

III. HISUI

Our Hisui consists of Hisui language specialized for image and video processing and Hisui compiler that performs advanced optimization on programs written in the language. Hisui language is a DSL with a high level of abstraction, which follows the specification of our previously proposed language [11] for image and video processing. In this section, we describe the features of Hisui language.

In general, image and video processing repeatedly applies certain processing to the components of an image, either the entire image or an arbitrary area. For example, the conversion of a color image to a monochrome image can be written in a general programming language as shown in Fig. 1(a). This code converts a 640 $\times$ 480 image stored as RGB values in a two-dimensional array "img" to grayscale. The processing data unit of this code is a pixel, and the process for each pixel is applied to all pixels. This code explicitly specifies the scope of processing in the loop statement.

Instead of a loop statement, Hisui language specifies a processing data unit and a processing data range. The code shown in Fig. 1(a) can be written in Hisui language as shown in Fig. 1(b), where "(pixel)p1" in the first line represents the processing data unit, and "(image)img1" represents the processing data range. Thus, the loop statement is substituted in the form "unit@range." In order to facilitate the writing of image and video processing programs, several data types are provided. The typical data types are shown in TABLE I. For example, the "pixel" type used in the first line of Fig. 1(b) represents a single pixel, and the "image" type represents a single image. Therefore, the "(pixel)p1@(image)img1" indicates the following three things.

- The processing for the data unit "p1" of the pixel type is defined later.

TABLE I
THE VARIETY OF DATA TYPES PROVIDED BY HISUI LANGUAGE.

type name	The object represented by the type
pixel	single pixel
box	partial image
image	image
stream	stream
array	versatile data array

- p1 represents an arbitrary data unit in "img1."
- The defined processing for "p1" is applied to all processing data units in "img1."

The first advantage of this notation is that various processing patterns can be expressed without major changes to the notation. Fig. 2 shows a video processing code that applies the image processing shown in Fig. 1(b) to each frame of a video. This code consists of the procedure for each image (lines 1-6) and for video (lines 7-11). In Hisui language, input/output variables for functions are defined by writing them adjacent to the function name via ">." Codes for video processing can also be written in the same notation as codes for image processing. This is because a video processing program is constructed by a set of image processing functions and rules to apply these functions to each image in a video.

The second advantage is that image and video processing programming can be abstracted. When writing image processing using a general programming language, it is necessary to specify the height and width of the image as shown in the first and second lines of Fig. 1(a) code. In addition, in the case of video processing, it is necessary to consider the frame rate. On the other hand, Hisui language provides a programming paradigm that does not require the user to be aware of such information. This enables intuitive programming at a high level of abstraction, thereby reducing the burden on programmers.

Finally, the third advantage is that the processing that can be parallelized and the data dependencies can be clear in the Hisui codes. When a loop statement is used as in Fig. 1(a), the iterator fixes the processing order inside the loop. However, such a fixed processing order often makes it difficult for a compiler to automatically parallelize the code even if the processing result does not change depending on the processing order. On the other hand, with Hisui language, the processing that can be executed in any order can be written without fixing

```
(image)img1 > Grayscale > (image)img2{
    (pixel)p1@img1{
        ave = (p1.R + p1.G + p1.B) / 3;
        p1.{R, G, B} = {ave, ave, ave};
    }
}
(stream)st1 > StreamGray > st1{
    (image)frame1@st1{
        frame1 > Grayscale > frame1;
    }
}
```

Fig. 2. A video processing code in Hisui language.

the processing order. As a result, the dependencies between the processing become clear, and Hisui compiler can easily find the processing that can be parallelized.

IV. Implementation

We implement a function that automatically performs the following three high-level optimizations on Hisui programs considering the processing patterns.

multi-threading
: divides a data range such as an image or a video into thread-count blocks, and assigns a thread to each data block.

vectorization
: applies SIMD instructions to handle multiple pixels simultaneously.

cache-blocking
: improves cache hit rate by dividing an image into sub-images, or blocks, to fit in cache and rearranging processing order to shorten data reuse time stride.

The Hisui compiler consists of a front-end Hisui translator, a back-end C++ optimizer, and a C++ compiler, as shown in Fig. 3. In this study, we implemented the C++ optimizer as an automatic optimization function in Hisui compiler. Currently, this feature is only available for x86 architecture.

As described in Sec. III, Hisui language uses the notation that specifies the processing data unit and processing data range. It is possible to express processing without specifying the order of processing, which has no ordering constraints. This makes it easy for Hisui compiler to extract parallel processing patterns from a source code written in Hisui language. To take advantage of this feature of this language, we have previously implemented a Hisui translator. This translator performs lexical and syntactic analysis on a source code, and then generates an intermediate code with annotation. The annotation indicates which part of the code can be parallelized. The C++ optimizer then performs advanced optimizations on the annotated intermediate code.

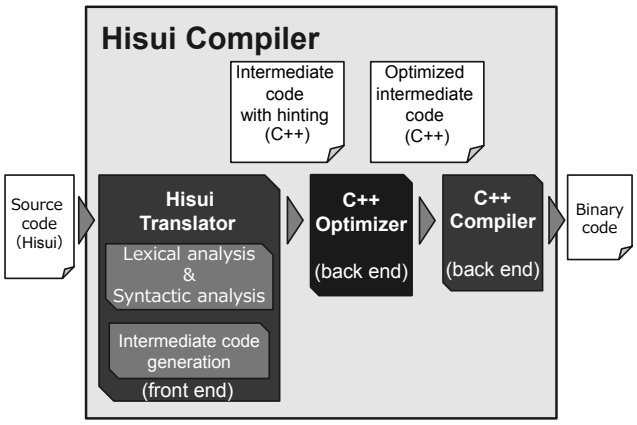


Fig. 3. Overview of Hisui compiler.

A. Multi-threading

In general, image and video processing programs have a high degree of parallelism, and many of them contain various parallel processing patterns [12], for example, maps and stencils. Map is a pattern that applies the same processing to all data units without any dependencies between them. Stencil is a pattern that uses the values of each data unit and its neighbors to update the value of the data unit.

The optimizer performs multi-threading on programs that contain these patterns. In multi-threading, dividing the area that can be processed in parallel into multiple blocks. Then, each block is processed in parallel by multiple threads to speed up the program.

The optimizer also provides the ability to bind each thread to a separate core, which is not provided in Halide. This feature prevents threads running on one core from being migrated to another core (thread migration). By preventing thread migration, the caches on each core can be used effectively, which is expected to improve program performance.

In the multi-threading survey, we applied multi-threading by single, half of the number of physical cores, the number of physical cores, the number of logical cores with the setup in order to determine how many threads have the greatest effect. We then measured the execution time with and without core-binding for each number of threads. We used grayscale transform and blur filter as workloads of maps and stencils, respectively. Also, four image sizes were used.

The result of the survey is shown in TABLE II, Fig. 4(a) and Fig. 4(b). In these figures, the vertical axis is the execution time ratio normalized to the processing of using a single thread, and the horizontal axis is the size of the image in pixels.

Both figures show that when the image size is larger than 2048×1200 pixels, multi-threading reduces the execution time and improves the performance. Also, when the image size is larger than 2048×1200 pixels, the execution time is shorter and the performance is higher when the thread is bound to the core. These results suggest that multi-threading with core-binding is effective for programs containing maps and stencils. Note that the overhead of multi-threading caused by thread creation and scheduling can degrade the performance as shown in the result with 1024×600 in Fig. 4(a). However, the results for all image sizes show that multi-threading by half of the number of the physical core with core-binding is faster on average.

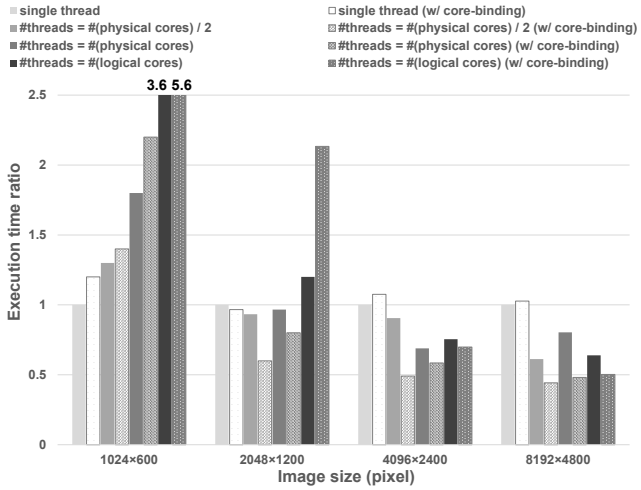
Therefore, we designed the optimizer to bind half of the number of physical cores threads to separate cores when applying multi-threading to programs containing maps and stencils.

B. Vectorization

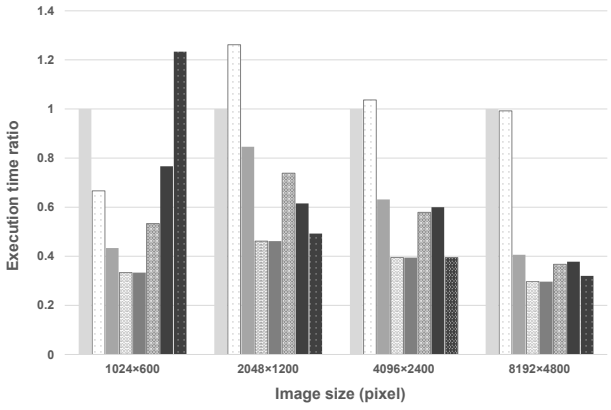
In addition to multi-threading, the optimizer vectorizes the programs containing maps and stencils. For vectorization, the optimizer uses SIMD instructions to process multiple pixels at a time to speed up the program.

TABLE II
SETUP WITH LARGE CACHE SIZE.

OS	Ubuntu 16.04
CPU	Intel Core i9-7900K
Clocks	3.3 GHz
Cache(L1d/L2/LLC)	32KB/1024KB/14080KB
Physical/Logical Core	10/20
Memory	64GB
Compiler	gcc 9.3.0
Compile options	-O3



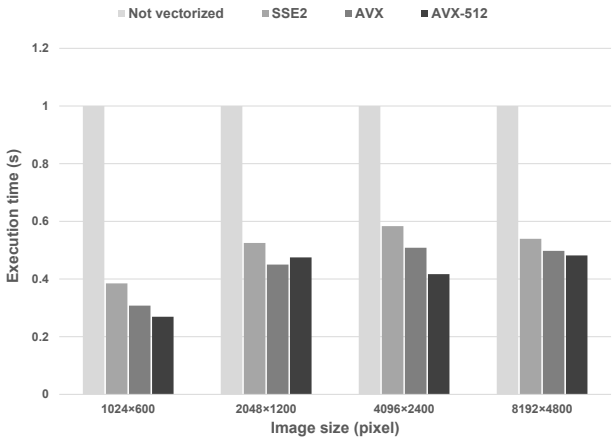
(a)Map



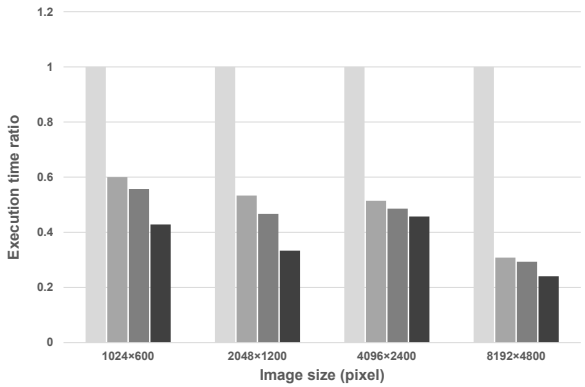
(b)Stencil

Fig. 4. Performance improvement by multi-threading.

For investigating the effect of vectorization, we used three SIMD registers, SSE2, AVX, and AVX-512[13] are 128, 256, and 512-bit width, in order to find out which instruction set is more effective. We measured the execution time without vectorization and when vectorization was applied using each SIMD register. In Fig. 4, the execution time normalized to the processing of without vectorization. The workload and image size used in the study are the same as those used in Sec. IV-A.



(a)Map



(b)Stencil

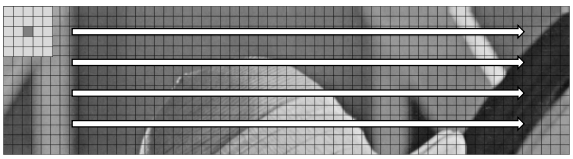
Fig. 5. Performance improvement by vectorization.

The result of the survey is shown in TABLE II, Fig. 5(a), and Fig. 5(b). In these figures, the vertical axis is the execution time ratio normalized to the time of without vectorization, and the horizontal axis is the size of the image in pixels. Both figures show that vectorization reduces the execution time of the program and improves performance. The figures also show that the performance differs depending on the SIMD instruction set. One of the reasons for this difference is that the register widths of SSE2, AVX, and AVX-512 are different. The wider the register, the larger the number of pixels that can be processed simultaneously.

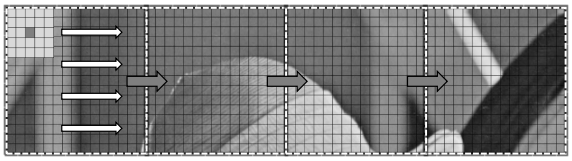
Therefore, we designed the optimizer to use the SIMD instruction set which can use the widest register among the SIMD instruction sets available on the platform.

C. Cache-blocking

In addition to multi-threading and vectorization, the optimizer applies cache blocking [14] to speed up the program. Reducing cache misses is important for performance improvement. Since the stencil in image and video processing refers to each pixel multiple times, cache misses frequently occur if



(a) A processing order without considering cache efficiency.



(b) A processing order considering cache efficiency.

Fig. 6. Changing the processing order by cache blocking.

TABLE III
SETUP WITH SMALL CACHE SIZE.

OS	Ubuntu 16.04
CPU	Intel Core i7-8700K
Clocks	3.7 GHz
Cache(L1d/L2/LLC)	32KB/256KB/12288KB
Physical/Logical Core	6/12
Memory	16GB
Compiler	gcc 9.3.0
Compile options	-O3

the cache is not effectively utilized as in Fig. 6(a). However, we can improve the cache hit ratio by changing the processing order considering the cache size as in Fig. 6(b).

The effect of cache blocking for a program containing stencils with the setup shown in TABLE III is shown in Fig. 7. In this figure, the vertical axis is the execution time and the horizontal axis is the size of the input image. A blur filter is used as the workload, and four types of input images are used. In this optimization, the processing order is changed by dividing the horizontal processing area into a certain size, as in Fig. 6(b). This fixed size is called the blocking size. In this study, the blocking size is set to be the width of the image divided by 2, based on the results of our preliminary evaluation.

Cache blocking will bring some overhead because the nested loops resulting from cache blocking require many conditional branch instructions to be executed. However, as shown in Fig. 7, when the image size is large enough, cache blocking decreases the program execution time and improves the performance. In order to improve the performance, cache blocking should be applied only in the case where more cache misses can be suppressed.

We designed the optimizer to apply cache blocking only to stencils and only when the size of the input image is larger than the cache capacity.

D. Parallelization for Reductions

We also implement automatic parallelization of reductions, which is one of the common processing patterns in image and video processing. Reduction aggregates multiple data units

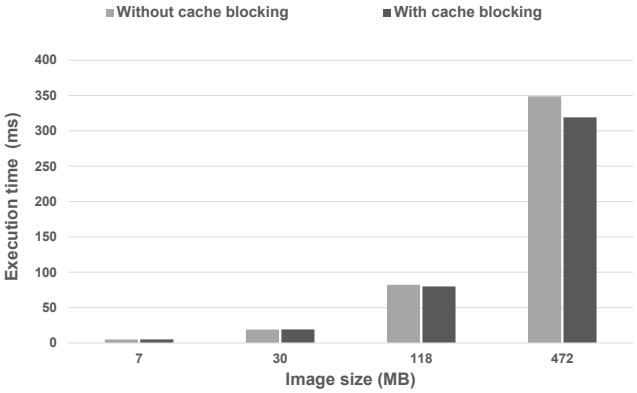


Fig. 7. Performance improvement by cache blocking.

```
1 int c = 0;
2 for (int i = 0; i < 6; i++) {
3   c = 2 * c + i;
4 }
```

Fig. 8. A code to update the value of variable c.

and combines them into a single data unit. In image and video processing, reductions are used when calculating the sum, maximum, histogram, and so on. Although parallelizing reductions is desirable, a previous work [15] shows that automatic parallelization of reduction is difficult to implement. In this study, we apply the methodology used in the existing study [16] by Jiang et al. that enables limited automatic parallelization of the reduction by analyzing the source code.

Let's see the methodology with a simple example code shown in Fig. 8. This is a code that updates the value of the variable c through operations using the value of itself, and the value of c given as input in each iteration depends on the result of the processing in the previous iteration. The six expressions shown in the upper part of Fig. 9 are the unrolled contents of the "for" statement in Fig. 8, and the value of the output c in the i-th (counting from 0) iteration is denoted as c_i. If the values of the input c_i in each iteration are all known, the calculations can be easily parallelized by multi-threading as shown in Fig. 9. However, the actual input c_i is determined depending on the processing of the previous iteration, the processing assigned to Thread 1 and 2 cannot be correctly parallelized with the processing of Thread 0.

To address this problem, Jiang et al. have proposed a method that enables limited automatic parallelization of the reduction by analyzing and reconstructing the contents of "for" statements. Here, we consider the following equation transformation for the processing of Thread 1 in Fig. 9.

$$\begin{aligned} c_3 &= 2 \times c_2 + 3 \\ &= 2 \times (2 \times c_1 + 2) + 3 \\ &= 2 \times 2 \times c_1 + 2 \times 2 + 3 \end{aligned} \tag{1}$$

Thus, c_3 can be expressed as a linear function of c_1. However,

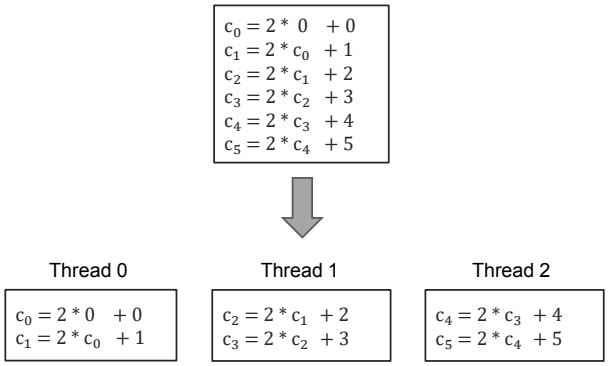


Fig. 9. Examples where parallelism is not applicable in practice.

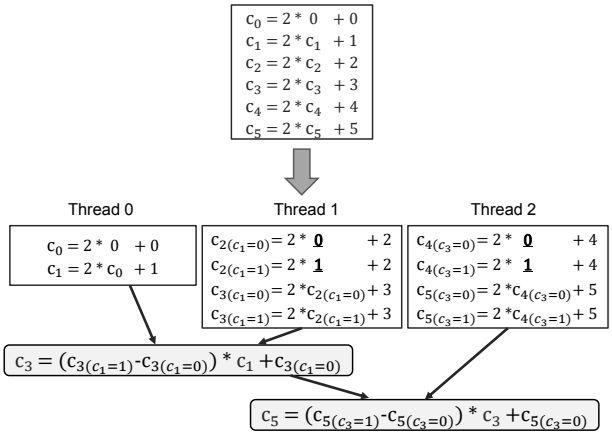


Fig. 10. Parallelization of the code shown in Fig. 8

c_1 will not be determined until the calculation by Thread 0 is completed. Therefore, the coefficient of c_1, 2×2, and the constant term, $2 \times 2 + 3$, are calculated first. First, since the coefficient of c_1 in Equation (1) corresponds to the slope of this linear equation, we write up c_3 as $c_{3(c_1=0)}$ when c_1 in Equation (1) is assumed as 0, and c_3 as $c_{3(c_1=1)}$ when c_1 is assumed as 1. Then, the slope of this linear equation can be obtained as $c_{3(c_1=1)} - c_{3(c_1=0)}$. The constant term $2 \times 2 + 3$ corresponds to the intercept in Equation (1), and can be obtained as $c_{3(c_1=0)}$. From these facts, Equation (1) can be generalized as follows.

$$c_3 = (c_{3(c_1=1)} - c_{3(c_1=0)}) \times c_1 + c_{3(c_1=0)}$$

Therefore, the slope and intercept of Equation (1) can be obtained by reorganizing the calculation with the values 0 and 1 as in Fig. 10 for c_1 of Thread 1 in Fig. 9. In the same way, we can calculate the slope and intercept of the equation to find c_5 in Thread 2 in Fig. 9.

In this manner, for a variable x that is updated in a loop, the value of x to be obtained in the j-th iteration, x_j, can be expressed in the following generalized expression using x_i such that $i < j$, and the parts corresponding to the slope and intercept can be calculated without waiting for the value of x_i to be determined if this is regarded as a linear expression for x_j.

$$x_j = (x_{j(x_i=1)} - x_{j(x_i=0)}) \times x_i + x_{j(x_i=0)}$$

The investigation setup of this method is the same as that used in Sec. IV-A, but core-binding is not applied. The reason for not applying core-binding is that reduction is not a parallel processing pattern that accesses the same address multiple times. We then measured the execution time for each number of threads. As a workload, we used a program that calculates the sum of the RGB values, which are the luminance values of the three primary color components, for all pixels of the input image. Four types of input images were used in the study.

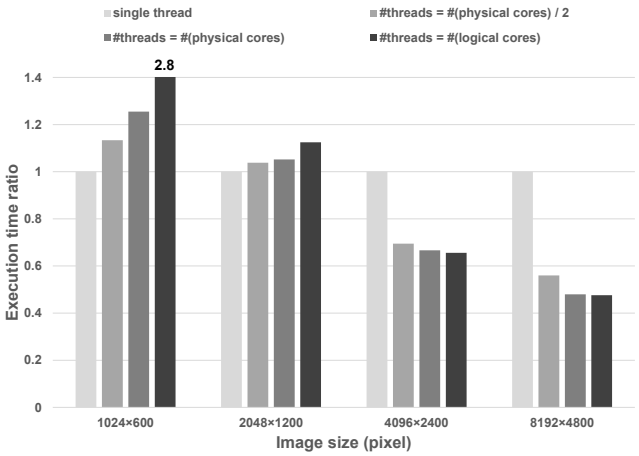


Fig. 11. Performance improvement by the optimizer: reduction.

The effect of this parallelization by the optimizer on the program containing the reduction is shown in Fig. 11. In this figure, the vertical axis shows the execution time ratio normalized to the processing of using a single thread, and the horizontal axis shows the size of the image in pixels.

As shown in Fig. 11, when the image size is larger than 2048×1200 pixels, applying this optimization reduces the execution time and improves the performance. Furthermore, with larger images, it is best for performance to use threads as the same number as physical cores. On the other hand, when the image size is less than 2048×1200 pixels, single-threaded processing is the best.

Therefore, we designed the optimizer to apply multi-threaded parallel reduction with threads as the same number as physical cores only when the input image size is larger than 2048×1200 pixels.

V. Evaluations

A. Experimental Environment

In this section, we evaluated the following two points in order to verify the usefulness of this framework.

- How easy to write in Hisui language.

- The performance of the programs generated by Hisui compiler, which implements the automatic optimization function.

As for the former, we compare the amount of codes between the codes written with OpenCV(C++), Halide, and Hisui language. As a workload, we used an image processing program that computes the sum of RGB values for all pixels in an input image.

As for the latter, we used the environments shown in TABLE II and TABLE III as the evaluation setup. We also used three image processing programs as workloads: a grayscale conversion, a blur filter, and a program to calculate the sum of RGB values for all pixels in the input image. In the case of grayscale or blur, we apply multi-threading by half the number of the physical cores with core-binding, and in the case of the sum of RGB, we apply multi-threading by the number of the physical cores. We measured the execution time of the programs optimized with -O3 option of the C++ compiler, the programs optimized by Halide with Auto-scheduler, and the programs optimized by Hisui compiler. These programs were executed 30 times each and the average execution times were compared. An image with 4096 $\times$ 2400 pixels was used as the input image.

B. Ease of Writing

Codes for calculating the sum of RGB values are shown in Fig. 12(a), Fig. 12(b), and Fig. 12(c).

As we can see in the figures, the source code written in Halide has the largest amount of code, while the source code written in Hisui is more concise than C++ and Halide due to its simple syntax described in Sec. III. One of the reasons why the code is more concise when using Hisui is that there is no need to specify the width and height of the image as described in Sec. III. On the other hand, the width and height of the image need to be specified in lines 3 and 4 of Fig. 12(a) in C++ and in line 12 of Fig. 12(b) in Halide. Another factor is that Hisui can concisely express the input and output of functions.

C. Performance

In order to verify the usefulness of Hisui, we evaluated the performance of programs generated by Hisui compiler that implements the automatic optimization described in the previous section.

The evaluation results are shown in Fig. 13. In this figure, the vertical axis shows the execution time ratio normalized to C++, and the horizontal axis shows the size of the image in pixels. The results show that the programs optimized by the optimizer of Hisui framework are about 2.3 times faster on average and about 3.0 times faster at maximum than the programs optimized by Halide Auto-scheduler. One of the reasons why the optimizer was able to generate programs with higher performance than Halide Auto-scheduler for grayscale transformation and blur filter programs is that the optimizer provides a function to bind threads to the core, which is not provided by Halide.

```
1  long sum(cv::Mat img){
2    int sum = 0;
3    for (int y = 0; y < img.rows; y++){
4      for (int x = 0; x < img.cols;x++){
5        cv::Vec3b *src =img.ptr<cv::Vec3b>(y);
6        sum += src[x][0] + src[x][1] + src[x][2];
7      }}
8    return sum;
9  }
10 int main(int argc, char *argv[]){
11   cv::Mat image = cv::imread(argv[1]);
12   long result = sum(image);
13   return 0;
14 }
```

(a) code in C++.

```
1  class AutoScheduled:public Halide::Generator<AutoScheduled> {
2  public:
3    Input<Buffer<uint8_t>> input{"input",3};
4    Output<Buffer<long>> sum{"sum",1};
5    void generate() {
6      RDom r(0, input.width(), 0, input.height(), 0, input.channels());
7      sum(x) = cast<long>(0);
8      sum(x) += input(r.x, r.y, r.z);
9    }
10   void schedule() {
11     if (auto_schedule){
12       input.set_estimates({{0,4096},{0,2400},{0,3}});
13       sum.set_estimates({{0, 1}});
14     }
15     else {
16       sum.compute_root();
17     }
18   }
19 private:
20   Var x{"x"}, y{"y"};
21 };
22 HALIDE_REGISTER_GENERATOR (AutoScheduled,auto_schedule_gen)
23
24 int main(int argc,char **argv){
25   Buffer<uint8_t> input = load_and_convert_image(argv[1]);
26   Buffer<long> sum(1);
27   auto_schedule_true(input,sum);
28   return 0;
29 }
```

(b) code in Halide.

```
1  (image)img1>sum>(long)out{
2    out=0;
3    (pixel)p1@img1{
4      out += p1.R + p1.G + p1.B;
5    }
6  }
7  (image)in>main>(long)out{
8    in > sum > out;
9  }
```

(c) code in Hisui language.

Fig. 12. Codes for calculating the sum of RGB.

To confirm the effect of binding to a core, we measured the number of thread migrations and L2 cache misses that occurred during the execution of the grayscale conversion program by using Linux perf command. The results are shown in TABLE IV. As shown in this table, the optimizer-optimized grayscale conversion program has fewer thread migrations and

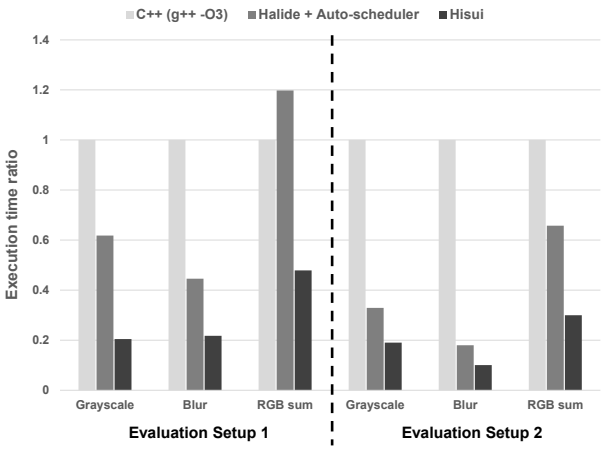


Fig. 13. Comparison of each workload.

TABLE IV
EFFECT OF CORE-BINDING: GRAYSCALE CONVERSION.

	Halide	Hisui
Number of thread migrations	59	20
L2 cache miss count	37,635	18,226
Run time	0.00618	0.00204

L2 cache misses than Halide, indicating that the cache is effectively utilized.

One of the reasons why the optimizer was able to generate a program with higher performance than Halide's Auto-scheduler in the program to calculate the sum of RGB values is that the optimization by the Auto-scheduler is not as appropriate as our Hisui optimizer. The optimized code derived by the Halide Auto-scheduler is multi-threaded, just like the Hisui optimized code, but also includes other modifications such as reordering the computation, that may bring some performance deterioration.

VI. CONCLUSION

In this paper, we have introduced an image/video processing framework Hisui and the initial implementation of its optimizer. As a result of evaluating the usefulness of our framework, we confirmed that the programs optimized by the optimizer are about 2.3 times faster on average and up to 3.0 times faster than those optimized by Halide's Auto-scheduler. Even programmers who are not familiar with computer architecture or parallel processing can write concise image and video processing programs by using Hisui language, and obtain high-performance programs by optimizing the programs with the dedicated compiler.

Future work includes installing other types of optimizations to Hisui optimizer. It is necessary to implement an automatic optimization function not only for image processing programs but also for video processing programs.

Another future challenge is to support various platforms such as GPUs, mobile processors, and ASICs. Currently, Hisui only supports x86, and our methodology may be optimized only to the x86 architecture. Therefore, the challenge is to make Hisui compatible with various platforms, not only to increase the portability of Hisui, but also to increase the versatility of the optimization method.

In addition, the optimizer should be able to handle various parallel processing patterns. Currently, the Hisui optimizer supports only map, stencil, and reduction among parallel processing patterns. However, image and video processing includes much more parallel processing patterns other than them.

REFERENCES

[1] Furuhashi, K., Noda, K., Ono, K. and Tsumura, T.: Highly Abstracted Video Processing Language Meets Multi-Grain Auto-Parallelization, *Proc. 12th Int'l Conf. on Signal-Image Technology and Internet-Based Systems (SITIS2016)*, pp. 127–134 (2016).

[2] Stewart, R.: An Image Processing Language: External and Shallow/Deep Embeddings, *Proc. 1st Int'l Workshop on Real World Domain Specific Languages (RWDSL'16)*, ACM (2016).

[3] Hegarty, J. et al.: Rigel: flexible multi-rate image processing hardware, *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2016*, ACM (2016).

[4] Korhonen, V. et al.: Rapid customization of image processors using Halide, *Proc. IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, IEEE, pp. 27–29 (2014).

[5] Hegarty, J. et al.: Darkroom: compiling high-level image processing code into hardware pipelines, *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2014*, ACM (2014).

[6] Mullapudi, R. T. et al.: PolyMage: Automatic Optimization for Image Processing Pipelines, *Proc. 20th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*, ACM, pp. 429–443 (2014).

[7] Baghdadi, R. et al.: Tiramisu: A polyhedral compiler for expressing fast and portable code, *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, IEEE Press, pp. 193–205 (2019).

[8] Ragan-Kelley, J. et al.: Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines, *ACM Transactions on Graphics (TOG) - SIGGRAPH 2012 Conference Proceedings*, ACM (2012).

[9] Adams, A. et al.: Learning to Optimize Halide with Tree Search and Random Programs, *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2019*, ACM (2019).

[10] Mullapudi, R. T., Adams, A., Sharlet, D., Ragan-Kelley, J. and Fatahalian, K.: Automatically Scheduling Halide Image Processing Pipelines, *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2016*, ACM (2016).

[11] Ono, A., Kondo, K., Inaba, T., Tsumura, T. and Matsuo, H.: A GPU-supported High-Level Programming Language for Image Processing, *Proc. 7th Int'l Conf. on Signal-Image Technology and Internet-Based Systems (SITIS2011)*, pp. 245–252 (2011).

[12] McCool, M., Reinders, J. and Robison, A.: *Structured parallel programming: patterns for efficient computation*, Elsevier (2012).

[13] Intel Corp.: Intel C++ Intrinsic Reference. http://software.intel.com/sites/default/files/a6/22/18072-347603.pdf.

[14] Lam, M. D. et al.: The cache performance and optimizations of blocked algorithms, *ACM SIGARCH Computer Architecture News*, ACM, pp. 63–74 (1991).

[15] Fisher, A. L. and Ghuloum, A. M.: Parallelizing complex scans and reductions, *ACM SIGPLAN Notices*, ACM, p. 135146 (1994).

[16] Peng.Jiang, Chen, L. and Agrawal, G.: Revealing parallel scans and reductions in recurrences through function reconstruction, PACT '18: Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, ACM, p. 113 (2018).