Highly Abstracted Video Processing Language Meets Multi-Grain Auto-Parallelization

Kazuki FURUHASHI*, Kensuke NODA*, Kazuma ONO* and Tomoaki TSUMURA*
*Nagoya Institute of Technology
Gokiso, Showa, Nagoya, Japan
Email: camp@matlab.nitech.ac.jp

***Abstract*—Multi-core and SIMD-supported processors have been popular and now can be highly utilized for image/video processing. However, to exploit theoretical performance of such a platform, developers must write programs, considering detailed specifications of the platform. We have proposed a video processing library RaVioli for solving this issue. RaVioli conceals two resolutions, frame rate and number of pixels, from developers for simplifying image/video processing programming. However, image/video processing programs could be written with more abstract syntax than that RaVioli provides, and image/video processing has much more potential parallelism than that RaVioli can exploit. To solve these problems, in this paper, we propose a multi-grain parallel video processing environment composed of a highly abstracted programming language and a dedicated compiler. The environment makes image/video processing programming easier and achieves higher performance.**

I. Introduction

Multimedia applications are now widely in demand, as many systems such as mobile computers and home information appliances have gained popularity. Such applications generally have data parallelism. Hence, SIMD instruction sets which most processors now support can be utilized, and have been constantly extended. Simultaneously, multi-core processors which also can exploit data parallelism in multimedia applications have become popular. The platforms which support SIMD and/or multi-thread on multi-core have much potential for high-performance image/video processing. However, to exploit theoretical performance of such a platform, developers must write and tune up programs, considering detailed specifications of the platform.

To address this problem, we have proposed a high-level video processing library *RaVioli* (Resolution-Adaptable Video and Image Operating Library) [1], [2], [3]. RaVioli conceals pixel rate and frame rate from developers, and can exclude the concept of resolutions from image/video processing programming. With this library, developers can write image/video processing programs more intuitively. RaVioli also partly supports auto-parallelization on some platforms. However, there still is room for improvement on RaVioli, from two points of view. One is that image/video processing programs could be written with more abstract syntax than that RaVioli supports. The other is that image/video processing has much more potential parallelism than that RaVioli can exploit.

These two issues are in the relationship of trade-off, because high abstraction of programming leads to large abstraction overhead, and minute performance tuning generally requires detailed designation of processing scheduling and/or hardware resource utilization. Through the development of RaVioli, we found that it is difficult to enhance both of programmability and performance by only a high-level library.

In this paper, we propose a novel video processing environment that provides both of a highly abstracted programming language and high performance with multi-grain auto-parallelization. The video processing environment also provides a dedicated compiler, that bridges the gap between programmability and performance by generating highly parallelized assembly codes automatically from the highly abstracted language.

II. Related Work

Many libraries such as OpenIP [4], OpenCV [5] have been developed for image/video processing [6], [7]. These libraries provide high-level writeability for image/video processing and they can parallelize image processing programs supported by other parallelization libraries such as OpenMP. However, in order to write parallel programs with these libraries, developers are required to have knowledge of not only image processing but also parallelization.

Some programming languages for image processing have also been proposed [8], [9], [10], [11]. A loop-less image processing language [9], for example, allows developers to implement image processing for embedded devices without any knowledge about the processors or memory architectures. Using this language, developers can write programs with some special iteration operators instead of loops. However, developers still have to consider resolutions even with this language.

Halide [10], [11] is the most well-known and promising programming framework specialized for image processing. In the aspects of the performance, readability, and writeability, Halide is highly appreciated compared with several previous image processing frameworks [12], and much work inspired by Halide is being conducted now [12], [13], [14], [15], [16], [17], [18]. Halide can support various platforms such as general-purpose multi-core processors, GPUs, and mobile processors. In Halide programs, the core algorithm for image processing and parallelization procedure are decoupled, and developers

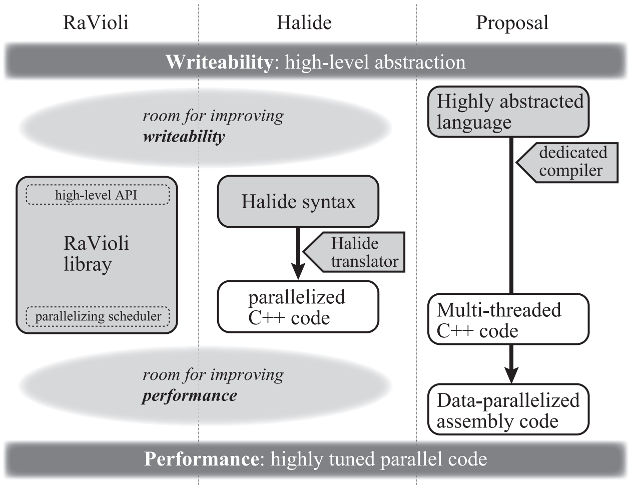
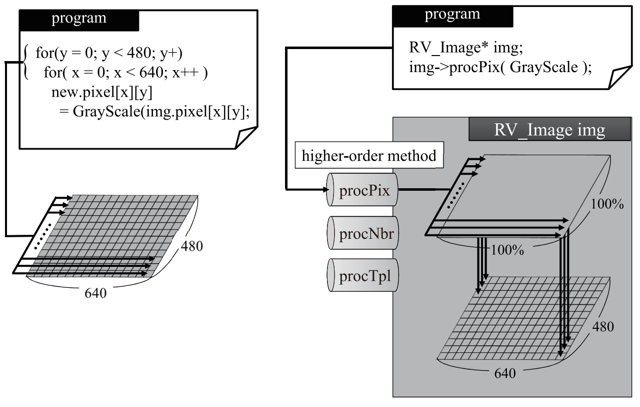

RaVioli
Halide
Proposal
Writeability: high-level abstraction
room for improving writeability
Highly abstracted language
dedicated compiler
high-level API
RaVioli library
parallelizing scheduler
Halide syntax
Halide translator
parallelized C++ code
Multi-threaded C++ code
room for improving performance
Data-parallelized assembly code
Performance: highly tuned parallel code

Fig. 1. Design concept overview. RaVioli and Halide still have room for improvement of programmability and performance. Highly abstracted video processing language and a dedicated compiler for the language that we propose can bridge the gap between programmability and performance.



```
for(y = 0; y < 480; y+)
  for( x = 0; x < 640; x++ )
    new.pixel[x][y]
      = GrayScale(img.pixel[x][y];
```

```
RV_Image* img;
img->procPix( GrayScale );
```

(a) A typical program. (b) A RaVioli program.

Fig. 2. Overview of digital image processing. In typical programs, kernel processing to a component, such as a pixel, is enclosed in nested loops. In RaVioli programs, a component function is passed to one of the higher-order methods that an image instance has.

can try several scheduling for parallelization without modifying the core algorithm for image processing. Halide also supports auto-parallelization for image processing programs. However, Halide does not natively support video processing. Hence, developers must write a video processing program as a combination of image processing for all frames, and control processing schedule for the frames.

The concept of Halide is very similar to that of RaVioli, and faces the same problem with RaVioli as shown in Fig. 1. Specifically, Halide has not yet coped with both high programmability and high performance. For example in Halide syntax, conditional statements must be described with some built-in functions. This is far from intuitive for developers because such a syntax is completely different from the common syntax using if-then-else statements. On the other hand, performance tuning with Halide is not well simplified. Developers still should select and use a suitable function for the processing from among many built-in parallel scheduling functions that Halide provides. In addition, developers should set appropriate parallelizing configuration parameters for the scheduling function. These are difficult for typical programmers.

In this paper, we propose a multi-grain parallel video processing environment. The environment can achieve high performance image/video processing with more simple syntax than the existing programming frameworks mentioned above provide. The environment also provides a dedicated compiler which generates multi-grain parallelized programs automatically. As a result, the environment makes image/video processing programming easier and achieves higher performance. We will compare the performance of the framework with Halide, and discuss advantages and disadvantages of them in Section VI.

III. Overview of RaVioli

Generally, loop iterations are heavily used in image/video processing programs. When converting a color image to grayscale, for example, each pixel will be converted to grayscale in the innermost iteration, and it is repeated for every pixel by nested loops. In typical image processing programs, developers have to know the height and the width of the image for defining the number of iterations of loops as shown in Fig. 2(a).

On the other hand, with RaVioli, an image is encapsulated in an RV_Image instance, and this repetitious processing for all pixels is applied by RaVioli automatically. Hence, developers should only write a routine for a component of the image and pass it to a higher-order method as shown in Fig. 2(b). In Fig. 2(b), GrayScale() is a routine defined by a developer. What developers should do are defining a function which processes one pixel and passing the function to one of the image instance's public methods. We call this type of function a *component function*. In this example, the component function GrayScale() is passed to procPix() which is defined as a higher-order method of the RV_Image instance. It applies the function passed as its argument to all pixels in the RV_Image instance one after another. Not only procPix(), RaVioli also provides some higher-order methods for several processing patterns [2]. This framework allows developers to be released from resolutions and the number of iterations. With RaVioli, it is easy to achieve auto-parallelization because the process for a component of an image is defined as the function. Therefore, RaVioli can support various platforms such as general-purpose multi-core processors, Cell/B.E., or CUDA GPUs [3].

However, there still is room for improvement on RaVioli, from two points of view. One is that image/video processing programs could be written with more abstract syntax than that RaVioli provides. When developers use RaVioli, developers still have to select the appropriate higher-order method for the processing pattern of applications. The other is that image/video processing has much more potential parallelism than that RaVioli can exploit. RaVioli can support various parallel processing platforms, but RaVioli cannot support multi-grain parallelization such as the vectorization, the tiling and the pipelining. RaVioli also causes the overhead for calling functions because processes for the component are defined as the function and they are repeatedly applied to each component.

IV. A Multi-Grain Parallel Video Processing Environment

In this section, we describe overview of the image/video processing environment we propose, and introduce specification of the image/video processing programming language. Then, we describe the multi-grain parallelization which is utilized in the environment.

A. Overview of the Image/Video Processing Environment

In this paper, we propose a multi-grain parallel video processing environment. The environment is composed of a programming language which provides higher abstraction of video processing than RaVioli and a compiler which generates a highly parallelized assembly code automatically. The video processing language is designed as an extension of the image processing language we have proposed [19]. The language provides an universal syntax for writing several processing patterns in the same format, unlike RaVioli.

The environment also provides a dedicated compiler for the language. The compiler can generate assembly programs which utilize multi-grain parallelism and exploit vectorization, tiling, and pipelining. As a result, with the environment, developers can highly derive the performance of the platform without any knowledge about the platforms and parallelization.

B. Highly Abstracted Video Processing Programming Language

The language provides a programming paradigm which conceals number of pixels and frame rate, like RaVioli. In addition, unlike RaVioli, developers can write programs without considering processing patterns, by using the universal syntax provided by this language.

1) Image Processing Function: Figure 3 shows a grayscale program written in the language. In this program, the video processing (line 7–11) and the image processing, or the processing for each frame (line 1–6) are written separately. Now, we explain how developers write image processing programs in the language. First, a function name should be defined. Each function definition should have input/output variables next to the function name with ">." In this program, the function Grayscale which has the input/output variable img1. To make image/video processing programming easier, the language provides some types: *pixel* for a pixel, *box* for a partial-image, *array* for an array of scalar values, and so on. If a variable is defined with no type declaration, it is regarded as a scholar variable. Developers can define the type of a variable by indicating between "(" and ")" just before the variable name, as shown in Fig. 3.

```
1  (image)img1 > Grayscale > img1{
2      (pixel)p1@img1{
3          ave = (p1.R + p1.G + p1.B) / 3;
4          p1.{R, G, B} = {ave, ave, ave};
5      }
6  }
7
8  (stream)st1 > StreamGray > st1{
9      (image)frame1@st1{
10         frame1 > Grayscale > frame1;
11     }
12 }
```

Fig. 3. Definitions of an image processing function 'Grayscale' and a video processing function 'StreamGray.'

```
1  (stream)in1 > main > (stream)out1{
2      in1 > Binary | Edge > out1;
3  }
```

Fig. 4. Definition of a main function. Multiple processing functions can be concatenated by "|," and treated as multiple stages for pipelining by the dedicated compiler.

Now, we explain how developers define the body of an image processing function (line 2–5). An **unit** variable being processed locates at the left side of "@," and an **area** variable where the unit processing is applied locates at the right side of "@." Namely, "(pixel)p1@img1" at line 2 in the figure means that "the *pixel* type variable p1 is the component unit of the *image* img1, and the process for the unit p1 is defined in the following block." Therefore, the process for p1, which is defined at line 3–4, is applied to all pixels in img1. At line 3, the average of each value of RGB of p1 is calculated, and at line 4, the average is assigned to each of RGB of p1, and p1 is converted into grayscale.

2) Video Processing Function: In video processing programs, a function for video processing need to be defined, in addition to the component function of the image processing for each frame. The variable type *stream* is provided for defining a video stream. Now, we explain how to write video processing programs in the language, by using the sample program shown in Fig. 3. As mentioned in Section IV-B1, Grayscale defined at line 1–6 is the function for converting an image into grayscale. Besides, the function StreamGray for video processing is defined at line 7–11, with a *stream* variable st1 as its input/output. In this function, the process for st1 is defined. The expression "(image)frame1@st1" at line 8 means that "the *image* type variable frame1 is the component unit of the *stream* st1, and the process for the unit frame1 is defined in the following block." In the definition block of StreamGray, Grayscale defined at line 1-6 is applied to all frames in the st1 (line 9). In this way, video processing programs can be written in as similar form as image processing programs.

3) Main Function: A program needs a main function in addition to image/video processing functions. As shown at line 1 in Fig. 4, a function defined with the name "main" is assumed as the main function, and will be called at the begin-

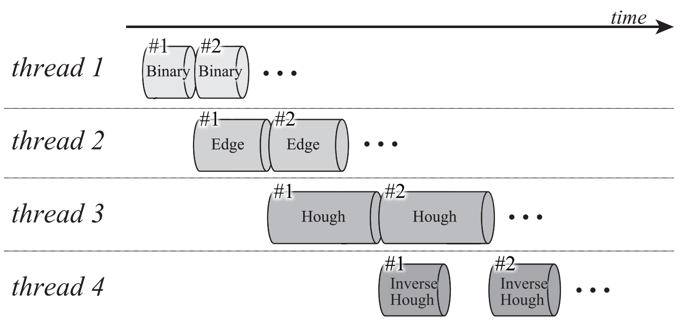


Fig. 5. Pipelined execution. Multiple processing functions defined as pipeline stages are overlapped by using multiple threads on cores. Load imbalance perceived in this figure is approximately resolved by the dedicated compiler as we describe in Section V-C and Fig. 9, later.

ning of the program. Input and output variables for the main function can be defined in the same format as image/video processing functions. The input/output variables are mapped to input/output videos respectively. In the case of image processing programs, the type for input/output variables of the main function should be *image*. Here, an image/video application is commonly composed of several processing phases. The symbol "|" can be used to enumerate the processing phases when writing such an application. In Fig. 4, the functions Binary and Edge are applied to each frame of the input stream in1 one after another, and the stream out1 is output as the result. Besides, the output of the function located before "|" is mapped to the input of the function located after "|." In this way, the language allows developers to write video processing programs which is composed of multiple phases, and supports various types of video processing.

C. Multi-Grain Parallelization

The video processing framework also provides a dedicated compiler, which generates a highly parallelized assembly program automatically from a program written in the language explained in Section IV-B. For the parallelization, we focus on two types of parallelism in image/video processing.

One is data parallelism. A data component in an image, such as a pixel or a pixel set, and another data component have data parallelism. Common processing on the components can be applied in parallel, while it is applied sequentially by loops in naive implementation with procedural languages. Such a component can be easily detected in the proposed language, as a left side variable of "@." Hence, the compiler automatically vectorize the processes on such components utilizing SIMD instructions.

In addition, tiling, or block decomposition, is also effective for data parallelism in image processing. When a processor is equipped with multiple cores, the compiler automatically divide an image into multiple blocks, and parallelize the image processing by running multiple threads on the multiple cores. Tiling can be used cooperatively with vectorization.

The other is task parallelism. Processing on different frames

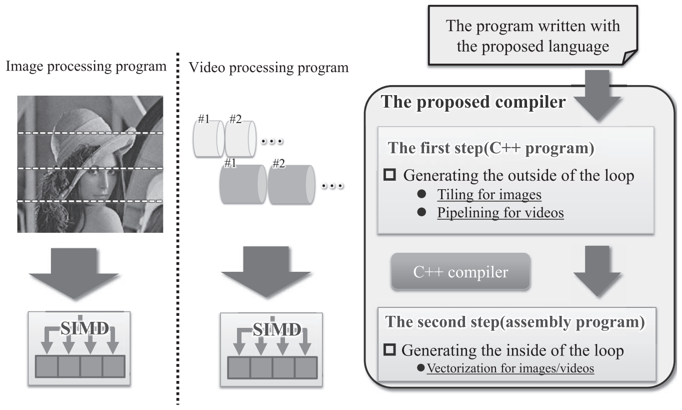

The program written with the proposed language
Image processing program
Video processing program
The proposed compiler
The first step(C++ program)
Generating the outside of the loop
Tiling for images
Pipelining for videos
C++ compiler
The second step(assembly program)
Generating the inside of the loop
Vectorization for images/videos
SIMD
#1
#2

Fig. 6. Overview of compilation. The dedicated compiler firstly applies multi-threading and generates thread-parallelized C++ code. Secondly, it applies data parallelization and generates assembly code filled with SIMD instructions. The compiler applies multi-threading by using block decomposition for image processing, and pipelining for video processing.

in a video stream has task parallelism. In addition, image/video processing applications often have several processing phases, and such phases can be indicated by using "|" in the proposed language, as mentioned in Section IV-B3. For such applications, pipelining the processing phases will be effective. For example, a typical straight line detection program is composed of four processing phases: binarization, edge detection, hough transform, and inverse hough transform. Hence, the program can be pipelined as shown in Fig. 5, by assuming the four processing phases as pipeline stages and mapping each stage to an individual thread on multi-core processors. In this figure, a number labeled at upper left of each processing stage shows a frame ID. As same as tiling, pipelining also can be used cooperatively with vectorization. Consequently, the video processing environment can exploit multi-grain parallelism in video processing applications.

V. Implementation of the Compiler

We implement a compiler which generates a highly parallelized assembly code from a program written in the proposed language. In this section, we describe the overview of the compiler. Then, we describe how the compiler generates an assembly code.

A. Overview of the Compiler

The compiler generates a highly parallelized assembly code through two steps as shown in Fig. 6. In the first step, the compiler applies tiling and pipelining to the program written in the proposed language. Specifically, the compiler translates only the outside of the kernel loops in image/video processing programs into C++ codes. Tiling and pipelining are implemented with POSIX threads. Then, the compiler translates the generated C++ program into an assembly program.

After that, in the second step, the compiler fills the inside of the kernel loops in the generated assembly codes with instructions which correspond to the processing for data units. At the time, the compiler uses SIMD instructions if possible,

```
1  int main(int argc,char* argv[]){
2    // Read an input image
3    Image* in = new Image();
4    IOHandler io;
5    io.Input(argv[1], in);
6    // Generate as many threads as the number of processor cores
7    for(int i=0; i<THRNUM; i++){
8      if(pthread_create(&thread[i], NULL, Grayscale, (void*)in) != 0)
9        exit(1);
10   }
11   // Join threads
12   for(int i=0; i<THRNUM; i++)
13     pthread_join(thread[i], NULL);
14   // Write an output image
15   io.Output(argv[2], in);
16   return 0;
17 }
```

Fig. 7. An example of a main function generated through the first step.

```
1  void* Grayscale(void* img1){
2    Image *img = static_cast<Image *>(img1);
3    int width = img−>getWidth();
4    int height = img−>getHeight();
5    // Acquire a thread ID
6    int mythreadID = getmyThrID();
7    // Calculate a region where the process is applied
8    int sj = (height / THRNUM) * mythreadID;
9    int ej = sj + (height / THRNUM);
10   if(mythreadID == THRNUM−1)
11     ej = height;
12   // Kernel loop
13   for(int j = sj; j < ej; j++){
14     for(int i = 0; i < width; i += CALC_WIDTH){
15       asm("nop");
16     }
17   }
18   return NULL;
19 }
```

Fig. 8. Grayscale function generated through the first step.

for vectorizing the program. In the following sections, we explain how the compiler generates highly parallelized assembly programs.

B. Tiling with Block Decomposition

First, the compiler analyzes the main function of a program written in the proposed language, and generates a main function in C++. Now, we explain how to generate the main function. Figure 7 illustrates an example code of a generated main function. The compiler generates codes for reading/writing images from/to input/output argument variables of the main function in the program written in the proposed language (line 3–5, 15). Then, in order to create threads each of which being allocated to individual processor core, the compiler inserts pthread_create() (line 7-10). At this time, the compiler creates as many threads as the number of cores. Incidentally, the number is acquired by profiling machine information in advance. Then, the compiler assigns each thread to a function by indicating it as an argument of the pthread_create(). In order to join threads which are allocated to processor cores after functions are executed, the compiler inserts pthread_join() (line 12–13).

After that, the compiler analyzes the component function of the program written in the proposed language, and generates a corresponding C++ function. Now, we explain how the C++ version of the component function Grayscale shown in Fig. 3 is generated. Here, Fig. 8 shows the generated code. Initially, the compiler generates codes for acquiring the width and the height of the input image (line 3–4). Then, in order to calculate which decomposed block of the input image should be assigned to each thread, the compiler generates a function for acquiring thread ID (line 6). If the number of threads is N, the value of ID can be from 0 to $N-1$. In addition, the compiler decides which block of an input image should be assigned to each thread by using the ID, the width of the image, the height of the image, and the number of threads (line 8–11). The image is horizontally divided considering spatial locality of reference, and each thread will access continuous memory area. At last, the compiler inserts kernel loops for processing each pixel in the image. In this step, the compiler inserts only a "nop" instruction inside the kernel loops by using inline assembly. This "nop" instruction works as a marker for finding where the kernel processing corresponds to the component function should be inserted in the second step later. Here, CALC_WIDTH, which is used in reinitialization of the kernel loop, shows the number of data which are processed in parallel by one SIMD instruction. As mentioned earlier, in the image processing program, the compiler generates codes only for tiling at the first step.

Now, if some shared variables are accessed in the component function, the accesses on the variables need exclusive control. In the program written in the proposed language, there is no data dependency between the processing on different components or data units even when some shared variables are accessed, because the processing order on the components is not defined unlike in the programs using loop structures. Hence, if some shared variables are detected in a component function, the compiler defines a thread-local variable for each shared variable. Then, the compiler generates codes for processing in parallel with reading and writing data on these thread-local variables, and generates codes for sequentially merging the results which are processed by each thread.

C. Pipelining

As tiling is applied to image processing programs, pipelining is applied to video processing programs by the compiler at the first step. First, the compiler analyzes the main function and the component function of a video processing program written in the proposed language, and generates a C++ main function. The compiler generates codes for reading/writing video from/to input/output argument variables of the main function. Second, the compiler examines whether the component function is defined as a series of multiple functions concatenated by "|." If it is composed of multiple functions, the compiler judges that those functions can be pipelined.

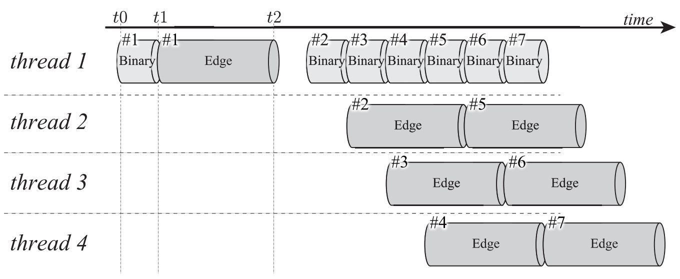


Fig. 9. An example of a pipelined execution flow. The number of threads binded to each processing stage is automatically adjusted according to the stage load.

Then, the compiler generates codes for calculating how many threads should be mapped to each function, based on the ratio of the processing load of the functions. After generating the main function, the compiler analyzes the component function for the image processing on each frame of the input video, and generates the corresponding C++ function.

We show how the execution flow of a video processing is pipelined by the compiler, using an example shown in Fig. 9. Now assume that this video processing program is composed of two processing stages: Binary and Edge. First of all, the two processing stages for the first frame is sequentially executed, and the time required for each stage is measured (Binary: $t1-t0$, Edge: $t2-t1$). By using this result, how many threads should be assigned to each function (or stage) is calculated for balancing each processing load. Specifically, the ratio of execution time for the processing stages is calculated, and the number of threads which should be assigned to each stage is calculated from the ratio. In Fig. 9, for example, the total number of threads is four, and the ratio of time of Binary to time of Edge is $1:3$. Thereby, the number of threads which are assigned for Binary is calculated as one and for Edge is three. Then, threads are created and assigned to each processing stage based on the calculated result. In this way, the second and the succeeding frames are processed in a pipelined manner as shown in Fig. 9.

D. Vectorization

In the second step, the compiler fill the codes inside the kernel loop, where a “nop” instruction is provisionally located in the first step. The program is vectorized in this second step with SIMD instructions. First, the compiler analyzes the assembly program generated through the first step, and searches “nop” instructions to detect kernel loops. After detecting a kernel loop, the compiler generates a code for filling inside the kernel loop. Figure 10 shows a part of filling code for the kernel of Grayscale shown in Fig. 3. In the filling code, the addresses of the pixels, which are being processed in the kernel, are stored into general registers. The pixels are components of the input image of the component function where the kernel loop locates. Then, the pixel data indicated by the addresses are moved from the memory into SIMD registers. Finally, the pixel data gathered into the SIMD registers are processed by SIMD instructions, and the results are moved back to the memory.

```
:
.L0:
vmovdqu (%r8), %xmm0
vmovdqu (%r9), %xmm1
vmovdqu (%r10), %xmm2
vpsllw $8, %xmm0, %xmm0
vpsllw $8, %xmm1, %xmm1
vpsllw $8, %xmm2, %xmm2
vpmulhuw .ONETHIRD(%rip), %xmm0, %xmm0
vpmulhuw .ONETHIRD(%rip), %xmm1, %xmm1
vpmulhuw .ONETHIRD(%rip), %xmm2, %xmm2
vpaddw %xmm0, %xmm1, %xmm0
vpaddw %xmm0, %xmm2, %xmm0
vmovdqu %xmm0, (%r8)
vmovdqu %xmm0, (%r9)
vmovdqu %xmm0, (%r10)
addl $8,        %eax
cmpl −12(%rbp), %eax
jne  .L0
addl $1, %eax
cmpl −8(%rbp), %eax
jne  .L0
:
```

Fig. 10. A part of filling code for the kernel of Grayscale.

TABLE I
EVALUATION ENVIRONMENT

OS	CentOS 6.4
CPU	Intel Core i7-4770
Clocks	3.4 GHz
Memory	16 GB
Cores	8
SIMD Register Size	256 bits
Compiler	gcc 4.8
Compile options	-O3

VI. EVALUATION

We evaluated the proposed video processing environment with some image/video processing programs. In this section, we compare the performance and writeability between the proposed video processing environment, RaVioli, and Halide.

A. Performance of Image Processing

We evaluated execution time of some image processing programs. The evaluation environment is shown in TABLE I. For this evaluation, we used a grayscale program, an emboss filter program, and an edge detection program. The spatial resolution of input images is XGA ($1,024 \times 768$ pixels).

The evaluation result is shown in Fig. 11. The execution time of each program is illustrated with four bars. The leftmost bar is the result of the program written in C++. The second is with RaVioli, the third is with Halide, and the rightmost is the result with the proposed video processing framework. Each bar is normalized to the execution time of C++ programs.

As shown in the figure, the programs with the proposed environment outperforms 9.1-fold on average and 16.4-fold at

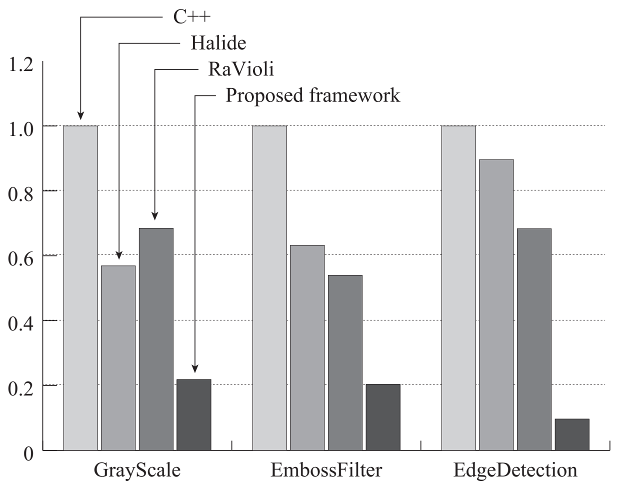


Fig. 11. Execution time of image processing programs. Halide and RaVioli achieve better performance than C++, with their auto-parallelization mechanism. However, they are inferior to the proposed framework because of their abstraction overhead.

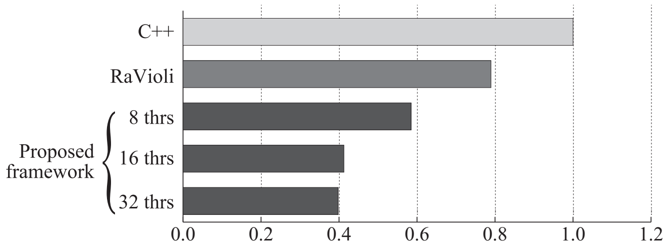


Fig. 12. Execution time of a video processing program. Halide does not natively support video processing, and its result is not shown. RaVioli provides good writeability also on video processing programs, but its abstraction overhead restricts its performance. The result with the proposed framework shows the performance advantage with pipelining, and the importance of processing scheduling across frames is proved.

a maximum against C++ programs. In addition compared with Halide, it outperforms 4.2-fold on average and 9.3-fold at a maximum.

With the proposed video processing environment, the execution time of all programs are reduced compared with RaVioli. Among the programs, the performance of the edge detection is especially improved. The program has multiple component functions, and large overhead of calling function costs with RaVioli. On the other hand with the proposed environment, the overhead does not cost much because the component functions can be inlined and vectorized by the dedicated compiler. Consequently, the proposed video processing environment can exploit much more performance of processors than existing other frameworks.

B. Performance of Video Processing

We also evaluated the execution time of a video processing program. For this evaluation, we used a straight line detection program composed of four processing stages: binarization, edge detection, hough transform, and inverse hough transform. The length of the input video stream is 30 seconds and the spatial resolution is XGA. This video stream is composed of 900 frames and the frame rate is 30 fps. The evaluation result is shown in Fig. 12. These bars in Fig. 12 show the execution time of programs with C++, RaVioli, and the proposed video processing environment. The performance with Halide is not evaluated because it does not support video processing. Horizontal axis in this figure indicates the execution time, and each bar is normalized to the execution time of the C++ program. Only pipelining and vectorization are applied to the program written in the proposed language, and tiling is not applied. The number of threads used for pipelined execution is 8, 16, and 32.

As shown in the figure, the program with the proposed environment outperforms 1.7-fold against the C++ program, and 1.4-fold against the RaVioli program. Here, in the straight line detection program, hough transform occupies more than 90% of the whole processing time for each frame. By examining how many threads are assigned to each processing, it is found that five threads were assigned to hough transform when the total number of threads is eight, while only one thread is assigned to each of the other processing stages. Hence, we have confirmed that the load is automatically balanced between the threads, and pipelined processing is fairly scheduled. When increasing the number of threads to 16, the performance is further improved because load is balanced more fairly by Hyper-Threading technology. However, the performance with 32 threads is not much superior to with 16 threads.

C. Comparison with Halide

In this section, we compare the proposed video processing environment with Halide from some points of view. Firstly, Halide can support various platforms, while the proposed environment currently supports only general-purpose multi-core processors. Thus, with Halide, developers can utilize many platforms for a program. However, with Halide, developers must use dedicated built-in functions for some types of expressions such as conditional branches, and the programming style is quite different from common procedural languages. Figure 13 and 14 show binarization programs written with Halide and the proposed video processing language, respectively. As shown in these figures, many dedicated built-in functions should be used with Halide, while the syntax of the proposed language is quite simple. Furthermore, some scheduling descriptions for parallelizing the program is required with Halide, as shown at line 12 in Fig. 13. On the other hand with the proposed environment, there needs no description for parallelization and the program is automatically parallelized. Next, with respect to the performance of image processing, the proposed environment clearly outperforms Halide as mentioned in Section VI-A. In addition, with respect to video processing, Halide originally does not support video processing, and developers must carefully design a video processing program, by using an image processing program written with Halide as a component of the video processing program and applying the component to each frame of the video with an appropriate processing scheduling. Consequently, the proposed environment can achieve higher performance with more simple

```
int main(int argc, char **argv) {
    Halide::Func Binary;
    Halide::Var x,y,c;
    Binary(x, y, c) = select( input(x, y, 0) < 85,
                              select( c == 0, input(x, y, 0) = 255,
                                      c == 1, input(x, y, 1) = 255,
                                      input(x, y, 2) = 255 ),
                              select( c == 0, input(x, y, 0) = 0,
                                      c == 1, input(x, y, 1) = 0,
                                      input(x, y, 2) = 0 ));

    Binary.vectorization(x, 16).parallel(y, 8);

    Halide::Image output = Binary.realize( input.width(), input.height(),
                                           input.channels());
}
```

Fig. 13. A binarization program written with Halide. As shown, developers should use select() functions for condition statements, and use a suitable parallel scheduling function with appropriate configuration parameters, in Halide programs.

```
(image)img1 > Binary > img1{
  (pixel)p1@img1{
    if(p1.R < 85) p1 = #black;
    else p1 = #white;
  }
}

(image)in > main > (image)out{
  in > Binary > out;
}
```

Fig. 14. A binarization program written in the proposed language. The syntax is more simple than Halide, and developers never need to consider parallelization.

description than Halide.

VII. Conclusion

In this paper, we proposed a video processing environment which can exploit several parallelism in programs. The environment is composed of a programming language which provides higher abstraction of video processing than existing languages, and a compiler which can generate a highly parallelized assembly program automatically. In order to confirm the usefulness of the proposed video processing environment, we compared the performance of the proposed environment with C++, RaVioli, and Halide by using some image/video processing programs. As a result, the programs written with the proposed environment outperforms the programs with any other languages and frameworks. In comparison to C++, the performance of the proposed framework is 16.4-fold at a maximum, and in comparison to Halide, it is 9.3-fold at a maximum.

One of our future work is to implement a mechanism for adjusting the total number of threads for video processing automatically and dynamically. When the load is severely imbalanced between processing stages, using more threads than the number of cores will improve the total performance. Another future work is supporting GPUs. It will be not so difficult because RaVioli has supported GPUs [3], [19]. Enhancing the writeability of the language, and increasing the number of video processing applications which can be thoroughly written with the language are also future work.

References

[1] S. Okada, H. Sakurai, T. Tsumura, and H. Matsuo, "RaVioli: A Resolution-Independent Video/Image Processing Library," *IPSJ Trans. on Computer Vision and Image Media*, vol. 2, no. 1, pp. 63–74, 2009, (in Japanese).

[2] H. Sakurai, M. Ohno, T. Tsumura, and H. Matsuo, "RaVioli: a Parallel Video Processing Library with Auto Resolution Adjustability," in *Proc. IADIS Int'l. Conf. Applied Computing*, vol. 1, Nov. 2009, pp. 321–329.

[3] K. Kondo, T. Inaba, H. Sakurai, M. Ohno, T. Tsumura, and H. Matsuo, "RaVioli: a GPU Supported High-Level Pseudo Real-time Video Processing Library," in *Proc. 19th Int'l Conf. on Computer Graphics, Visualization and Computer Vision (WSCG2011)*, Jan. 2011, pp. 39–48.

[4] G. Kovács, J. I. Iván, Árpád Pányik, and A. Fazekas, "The openIP Open Source Image Processing Library," in *Proc. Int'l Conf. on Multimedia (MM'10)*. ACM, 2010, pp. 1489–1492.

[5] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision With the OpenCV Library*. O'Reilly & Associates Inc, 2008.

[6] U. Köthe, "Generic programming for computer vision: The vigra computer vision library," http://hci.iwr.uni-heidelberg.de/vigra/, Sep. 2011.

[7] "Pandore: A library of image processing operators (Version 6.4). [Software]. Greyc Laboratory," http://www.greyc.ensicaen.fr/˜regis/Pandore, 2011.

[8] S. Wang, Z. Dong, J. X. Chen, and R. S. Ledley, "PPL: A whole-image processing language," *Computer Languages, Systems & Structures*, vol. 34, pp. 18–24, Apr. 2008.

[9] J. Segawa and T. Kanai, "The Array Processing Language and the Parallel Execution Method for Multicore Platforms," *Proc. 1st Int'l Symp. on Information and Computer Elements*, 2007.

[10] J. Ragan-Kelley, A. Adams, S. Paris, M. Leboy, S. Amarasinghe, and F. Durand, "Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines," in *ACM Transactions on Graphics (TOG) - SIGGRAPH 2012 Conference Proceedings*. ACM, Jul. 2012.

[11] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. P. Amarasinghe., "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proc. 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'13)*. ACM, 2013, pp. 519–530.

[12] R. Stewart, "An image processing language: External and shallow/deep embeddings," in *Proc. 1st Int'l Workshop on Real World Domain Specific Languages (RWDSL'16)*. ACM, 2016.

[13] J. Hegarty, R. Daly, Z. DeVito, J. Ragan-Kelley, M. Horowitz, and P. Hanrahan, "Rigel: flexible multi-rate image processing hardware," in *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2016*. ACM, Jul. 2016.

[14] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, and W. Eckert, "Hipacc: A domain-specific language and compiler for image processing," in *IEEE Trans. on Parallel and Distributed Systems*, vol. 27. IEEE, Jan. 2016, pp. 210–224.

[15] V. Korhonen, P. Jaaskelainen, M. Koskela, T. Viitanen, and J. Takala, "Rapid customization of image processors using halide," in *Proc. IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. IEEE, 2014, pp. 27–29.

[16] 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," in *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2014*. ACM, Jul. 2014.

[17] C. Thirumoorthi and T. Karthikeyan, "Easy optimization of image transformation using sfft algorithm with halide language," in *Int'l Conf. on Contemporary Computing and Informatics (IC3I)*. IEEE, Jul. 2014.

[18] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "Polymage: Automatic optimization for image processing pipelines," in *Proc. 20th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. ACM, 2014, pp. 429–443.

[19] A. Ono, K. Kondo, T. Inaba, T. Tsumura, and H. Matsuo, "A gpu-supported high-level programming language for image processing," in *Proc. 7th Int'l Conf. on Signal-Image Technology and Internet-Based Systems (SITIS2011)*, Nov. 2011, pp. 245–252.