This is the accepted manuscript of a paper published in
Proc. 2nd Int'l Symp. on Computing and Networking (CANDAR'14), pp.185-191

Auto-Parallelization for a Video Processing Library with Content-Aware Resolution Management

Masahiro MIZUNO*, Takuya MATSUNAGA*, Tomoaki TSUMURA* and Hiroshi MATSUO*
*Nagoya Institute of Technology
Gokiso, Showa, Nagoya, Japan
Email: camp@matlab.nitech.ac.jp

***Abstract*—The performance of general purpose computers is increasing rapidly, and now they are capable of running video processing applications. However, on general purpose operating systems, real-time video processing is still difficult because there is no guarantee that enough CPU resources can surely be provided. A pseudo real-time video processing library RaVioli has been proposed for solving this issue. RaVioli conceals two resolutions, frame rate and number of pixels, from programmers and provides a dynamic and transparent resolution adjustability. Namely, RaVioli regulates the processing load by automatically modifying two resolutions according to CPU utilization. Besides, RaVioli can divide a whole video frame into several sub-frames, and process each sub-frame in different precisions considering its importance. Using RaVioli, pseudo real-time video processing can be achieved easily, but output precision may be extremely roughened for reducing processing load even if each sub-frame is processed with the suitable precision. To solve this problem, we propose an auto-parallelization mechanism on RaVioli. We aim to improve the processing precision with the mechanism.**

I. Introduction

Real-time video processing applications such as surveillance systems, smoke detection systems or automatic vehicle collision avoidance systems are now in demand. On the other hand, the processors for general-purpose computers have been developed drastically. The performance of the processors has been improved, and the cost has been reduced. Therefore, it is also expected that the performance improvement and the cost reduction will promote real-time video processing on the general-purpose computers and operating systems. However, it is still difficult to realize the real-time video processing on general-purpose systems. The main reasons for the difficulty are the fluctuations in the computation load of each frame and in the amount of the available CPU resources.

To solve this problem, we have proposed a high-level video processing library *RaVioli* (Resolution-Adaptable Video and Image Operating Library)[1], [2] which guarantees real-time processing on general-purpose computing systems. RaVioli can regulate the processing load by automatically modifying spatial resolution and frame rate according to CPU utilization.

For such dynamic modification of the resolutions, a programming fashion which is independent of the resolutions is required. RaVioli conceals two resolutions, spatial resolution (i.e. pixel rate) and temporal resolution (i.e. frame rate), from programmers for automatically changing the resolutions at run-time. This can exclude the concept of resolutions from video processing programming, and developers can write video processing programs more intuitively.

Besides, RaVioli can divide a whole video frame into several sub-frames, and process each sub-frame in different precisions considering its importance[2]. This can keep the resolutions for important areas high.

However, the output precision may be extremely roughened for reducing processing load even if each sub-frame is processed with suitable precision. It is inevitable that the output precision is somewhat reduced for achieving realtimeness, but the precision should be kept as high as possible.

In this paper, we propose an auto-parallelization mechanism on RaVioli. We aim to improve the processing precision with the mechanism.

II. Related Work

A. Real-Time Video Processing

So far, several real-time video processing applications have been developed. For example, Garcia-Martin et al.[3] have presented a moving people detection for surveillance video systems. Kim et al.[4] have proposed a method for early smoke detection. Lin et al.[5] have presented a real-time eye detection algorithm.

As described in section I, it is difficult to implement real-time video processing applications on general-purpose systems. This is because that the amount of the available CPU resources or computation load can fluctuate. To solve this problem, some methods for adjusting the processing load have been proposed. Writing multiple routines with different algorithms has been the most-used solution for the load adjustment. One example is the imprecise computation model (ICM)[6]. In this model, computation accuracy varies corresponding to the given computation time limit. With the confidence-driven architecture [7], which is based on ICM, developers have to troublesomely implement multiple routines with different algorithms and different loads, and the confidence-driven architecture selects suitable routine dynamically and empirically among them. On the other hand, RaVioli can apply load-adjustment to any video processing applications automatically.

B. Libraries and Programming Languages

Many image/video processing libraries have been also developed. Adopting template techniques similar to the C++

STL, VIGRA[8] allows developers to easily adapt given components to their programs. OpenCV[9] provides many typical image/video processing algorithms as C functions or C++ methods. OpenIP[10] provides a set of interoperable, open source libraries, satisfying the demands of image processing and computer vision in education, research and industry, as well. Pandore[11] provides a set of executable image processing operators. It is dedicated to image processing experts because skills on image processing operations are needed to use this library. These libraries provide high-level descriptivity of image/video processing, but adjusting computation load is difficult to be implemented with them.

Some programming languages for image processing also have been proposed. A loopless image processing language[12], for example, allows developers to implement image processing for embedded devices without any knowledge about the processors or memory architectures. With this language, developers can operate arrays without using loops in programs with some special operators. However, developers have to write programs with a formula editor and consider array sizes.

Halide[13] is another programming language specialized for image processing. In Halide programs, the algorithm for image processing and parallelization procedure are separated, and developers can try several scheduling for parallelization without modifying the core algorithm for image processing. However, this means that Halide requires developers to have knowledge of not only image processing but also efficient parallelization.

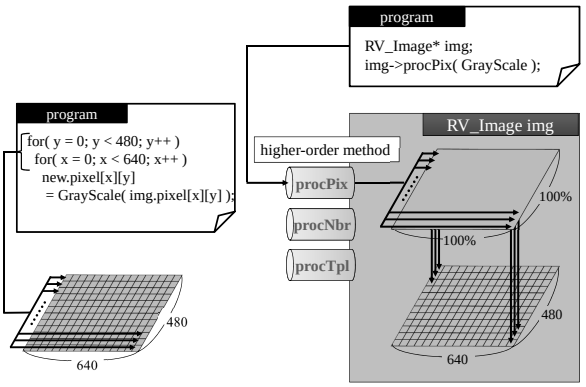
The approach of the library RaVioli[1], [2] is completely different from such existing computation models or image/video processing libraries and languages. RaVioli allows programmers to be unaware of the existence of pixels and frames through their video processing programming. Concealing pixels and frames from programmers, RaVioli can change spatial/temporal resolutions and can adjust processing load dynamically and automatically. RaVioli also can support various platforms such as Cell/B.E. or CUDA GPUs.

III. Overview of RaVioli

A. Abstraction of Image and Video Processing

Two resolutions, spatial resolution and temporal resolution of a video, are derived from the necessity of quantization on computers, and are not natural. We human beings naturally have no concept of resolutions through our visual recognition. For example, even though we can recognize object motion in our view without any pixel or frame, developers should consider resolutions for implementing a motion object detection application on computer systems. Hence, the presence of resolutions makes video processing programs unintuitive.

To solve this problem, RaVioli provides a new programming paradigm which conceals two resolutions spatial resolution and temporal resolution, from programmers. Hence, with RaVioli, developers can implement video processing applications without considering pixels and frames. Developers also can easily implement real-time video processing applications



(a) Traditional program. (b) Program with RaVioli.

Fig. 1. Digital image processing.

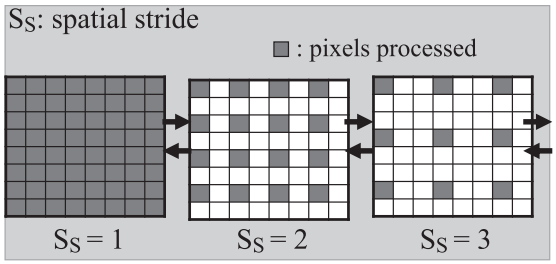
because RaVioli can automatically vary resolutions for adjusting computation load and achieving realtimeness.

Generally, loop iterations are heavily used in video processing programs. When converting a color image to grayscale, for example, each pixel will be converted to grayscale in the innermost iteration, and the process is repeated for every pixel by loop nests as shown in Fig. 1(a). In the case of neighborhood processing such as blur or edge enhancement, the processed unit is a set of a pixel and its neighbor pixels, and in the case of template matching, the processed unit is a small window in each frame. These units are processed in the innermost iteration, and the process is applied repeatedly by loop structures.

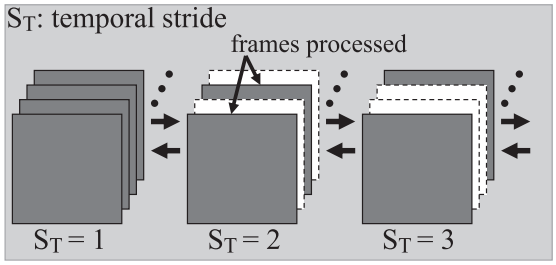
For using a double loop structure for image processing, programmers should know the height and the width of the image for defining the number of iterations of loops. On the other hand, with RaVioli, an image is encapsulated in an RV_Image instance, and this repetitious processing for all pixels is applied by RaVioli automatically, so developers should only write a routine for one pixel as shown in Fig. 1(b).

GrayScale() in Fig. 1(b) 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 class. It applies a function passed as its argument to all pixels in the RV_Image instance one after another. This framework allows developers to be released from resolutions and the number of iterations.

Not only procPix(), RaVioli also provides some higher-order methods for several processing patterns; such as template matching, k-neighbor processing, and so on. As same as an image, a video is also encapsulated in an RV_Streaming instance in RaVioli. Frames, the components of an RV_Streaming instance, are concealed from developers. An RV_Streaming instance also has several higher-order methods. Developers should only define a *component function*, which manages one frame, and pass the function to an appropriate higher-order

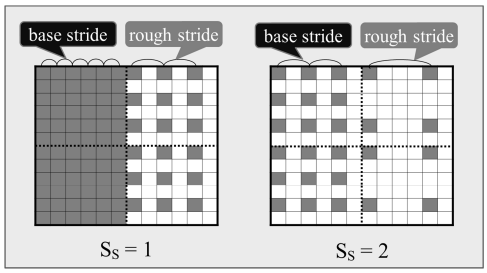


(a) Spatial stride and resolution.

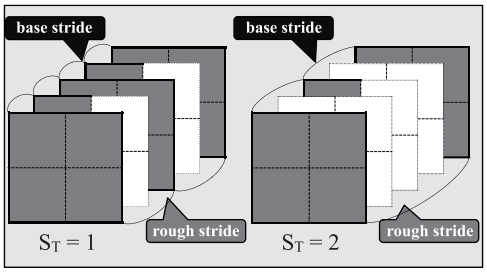


(b) Temporal stride and resolution.

Fig. 2. Changing processing resolution by stride access.



(a) Base and rough spatial strides.



(b) Base and rough temporal strides.

Fig. 3. Base stride and rough stride.

method of an RV_Streaming instance for video processing.

B. Real-Time Processing by Adjusting Computation Load

1) Resolution Adjustment: On general purpose systems, multiple processes are running concurrently. Hence, it is difficult to implement real-time video processing applications on such systems, because there is no guarantee that enough CPU resources can surely be provided. One solution for this problem is reducing computation load by roughly processing the video data, or reducing the resolution of the video. RaVioli can change spatial and temporal resolutions dynamically and automatically according to the available CPU resources, because RaVioli conceals resolutions from programmers.

RaVioli has two internal parameters; *spatial stride* (S_S) and *temporal stride* (S_T), and RaVioli changes processing resolutions by varying these parameters. Fig. 2(a) shows the relation between the value of spatial stride and the processing spatial resolution. Initially the value of spatial stride $S_S = 1$, and all pixels in each frame are processed. When the value of spatial stride is increased to $S_S = 2$, every other pixel is processed and the processing spatial resolution is roughened, and the whole processing load is reduced to $1/4$. For reducing the load more, spatial stride should be increased to $S_S = 3$ and the load is reduced to $1/9$.

Likewise, the relation between the value of temporal stride and the temporal resolution is shown in Fig. 2(b). Initially, the value of temporal stride $S_T = 1$ and all frames are processed. When the value of temporal stride is increased to $S_T = 2$, the temporal resolution is reduced by processing every other frame, and the whole computation load is reduced to $1/2$. If the value of temporal stride becomes $S_T = 3$, the whole processing load is reduced to $1/3$. As mentioned above, RaVioli can automatically vary resolutions for adjusting computation load and achieving realtimeness.

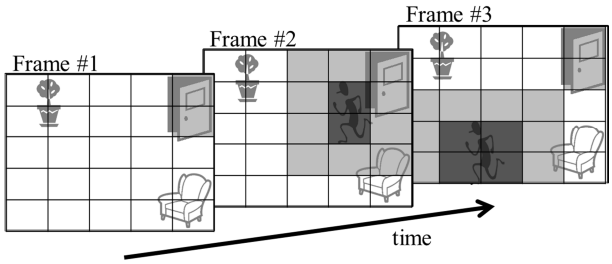


Fig. 4. An example of locating medium stride areas.

2) Content-Aware Resolution Management: In real-time video processing, some part of a frame will be an important area and should be processed precisely, but other part is not necessary to be processed precisely. From this perspective, RaVioli provides the content-aware resolution management[2]. Specifically, RaVioli can divide a whole video frame into several sub-frames, and process each sub-frame in different precisions considering its importance. For achieving the high processing precision of the important sub-frame, multiple options of spatial/temporal stride is provided for each video sub-frame. RaVioli provides three spatial/temporal strides; *base stride*, *rough stride*, and *medium stride*.

Base stride is a stride option for important areas. Important areas should be processed as precise as possible, and the value of base stride is set as small as possible.

On the other hand, *rough stride* is a stride for unimportant areas. Unimportant areas should be processed in low resolution to reduce whole processing load for achieving real-time video processing. Hence, the value of rough stride is defined as larger than base stride.

Now, Fig. 3(a) and Fig. 3(b) shows how base stride and

rough stride are applied to spatial and temporal resolutions. In both figures, a video stream is divided into 2×2 sub-streams, and the boundaries are depicted by dashed lines. First, if spatial base stride $S_S = 1$, unimportant sub-frames are processed with rough spatial stride 2, as shown in the left side of Fig. 3(a). In this case, the whole processing load of a frame is reduced to $5/8$, compared with the load when the whole frame is processed with $S_S = 1$. If $S_S = 2$, the value of rough stride is 4, as shown in the right side of Fig. 3(a). In this case, the whole processing load of a frame is reduced to $5/32$.

On the other hand, if temporal base stride $S_T = 1$, unimportant sub-streams are processed with rough temporal stride 2, as shown in the left side of Fig. 3(b). In other words, the unimportant areas in one of the each two frames will not be processed and the result of the previous frame is used. In this case, the whole processing load of this video is reduced to $3/4$. If base temporal stride S_T becomes 2, rough temporal stride becomes 4, as shown in the right side of Fig. 3(b). In this case, the whole processing load is reduced to $3/8$.

Next, *medium stride* is a stride option for *preferential areas*. The areas which may become important soon can be defined as preferential areas, and the resolutions of them are kept relatively high. Therefore, the value of medium stride is set between the value of base stride and the value of rough stride. This means that, an area processed with medium stride can reduce computation load compared with an area processed with base stride, and can detect changes rapidly in input videos compared with an area processed with rough stride.

Incidentally, we regard the areas where some change appears as important areas for realtime video processing. The location of important areas will vary because the contents of input video frames are changing every moment. However, there are some characteristics of such important input sub-frames. Considering these characteristics, preferential areas can be defined manually and automatically.

Now, let us see how one of the three strides is selected for each sub-frame in Fig. 4. In this example, assume that a surveillance system is implemented with RaVioli. In this figure, input frame are divided into sub-frames, and division is specified as 5×5. The number of sub-frames can be defined manually by programmers. The black-colored sub-frame means that the sub-frame is processed with base stride. Similarly, the gray and white-colored sub-frame means that they are processed with medium and rough stride respectively. The two sub-frames which cover the door are manually defined as preferential areas, because it is predictable that these areas are changeable. In the first frame, there is no change in the input, and the two preferential sub-frames use medium stride, and the other sub-frames use rough stride. In the second frame, an intruder appears from the door, and the input changes. Hence, base stride is applied to the sub-frames on the intruder. Then, the sub-frames around the current important sub-frames are automatically defined as preferential areas, because the important area will be located around the current important area soon. After that, if the intruder moves, appropriate strides are applied to all sub-frames according to the input as shown in the third frame in Fig. 4.

C. Problem with RaVioli

As mentioned above, RaVioli can keep the resolutions for important areas high. However, output precision may be extremely roughened for reducing processing load even if the suitable importance is set to each sub-frame. It is inevitable that the output precision is somewhat reduced for achieving realtimeness, but the precision should be kept as high as possible.

In this paper, to restrain a deterioration of the output precision, we propose an auto-parallelization mechanism on RaVioli. Using the mechanism, a sequential RaVioli program can be automatically translated into a parallel program. The mechanism also provides a load balancing algorithm which is suitable for the content-aware resolution management. We aim to improve the processing precision with the auto-parallelization mechanism.

IV. Auto-Parallelization

In this section, we will describe a preprocessor which automatically translates a sequential RaVioli program into a parallel program. Then, we will also describe the problem of the preprocessor and an improvement of the preprocessor.

A. Preprocessor for Auto-Parallelization

Now, multi/many-core processors have come into wide use. It becomes, therefore, much important to improve compilers and libraries for multicore processors being fully utilized.

Generally, loop iterations are heavily used in image processing programs and the loop body is applied to the units, such as pixels, repeatedly by loop structures as mentioned in section III-A. The loop iterations can run in parallel by utilizing data parallelism unless the loop has data dependency between iterations. However, parallelizing a loop can cause access conflicts on shared variables. To avoid such conflicts, *reduction operations* are generally used. Using reduction operations, each loop iteration can be executed completely independently.

For parallelizing image processing programs, programmers must deal with data dependencies between loop iterations, or access conflicts on global variables. To solve this problem, we have proposed a preprocessor [1] which automatically translates a sequential RaVioli program into a parallel program.

In an image processing program using RaVioli, some component functions are defined and they are applied to image instances through higher-order methods as described in section III-A. Incidentally, loop iterations which programmers usually write by themselves are now absorbed into RaVioli. Therefore, in a program using RaVioli, the preprocessor can detect parallelism easily and can conceal parallel processing from programmers. However, as mentioned above, access conflicts on shared variables may appear when executing RaVioli programs. To avoid such conflicts, when the previous preprocessor detects a global variable which satisfies the following both criteria, it decides that the variable needs a reduction operation.

sequential program

```
1.  int min=INT_MAX, sum=0;
2.  RV_Coord tmps, tmpe;
3.
4.  void Compare(RV_Pixel *p1, RV_Pixel p2){
5.    int r1, g1, b1, r2, g2, b2;
6.    p1->getRGB(r1, g1, b1);
7.    p2.get(r2, g2, b2);
8.    sum += absd(r1-r2)+absd(g1-g2)+absd(b1-b2);
9.  }
10.
11.  void countTP(RV_DoppImg* window, RV_Coord Cs,RV_Coord Ce){
12.    window->procImgComp(Compare, input_tp); //calling higher-order method
13.    if(min > sum){
14.      min = sum;
15.      tmps = Cs;
16.      tmpe = Ce;
17.    }
18.    sum = 0;
19.  }
20.
21.   int main(int argc, char* argc[]){
22.     RV_Image* input_img;
23.     input_img = new RV_img();  //input an image
24.     input_img ->procBox(CountTP); //calling higher-order method
25.     return 0;
26.     }
```

Fig. 5. Template matching program before translation.

parallel program

```
1.  int min=INT_MAX, sum=0;
2.  __thread int __min=INT _ MAX, __sum=0;
3.  RV_Coord tmps,tmpe;
4.  RV_Coord __tmps[4],__tmpe[4];
5.
6.  void Compare(RV_Pixel *p1, RV_Pixel p2){
7.    int r1, g1, b1, r2, g2, b2;
8.    p1->getRGB(r1, g1, b1);
9.    p2.get(r2, g2, b2);
10.   __sum+=absd(r1-r2)+absd(g1-g2)+absd(b1-b2);
11.  }
12.
13.  void CountTP(RV_DoppImg* window,  RV_Coord Cs, RV_Coord Ce){
14.    int thID = omp_get_thread_num();
15.    window->procImgComp(Compare, input_tp); //calling higher-order method
16.    if(__min > __sum){
17.      __min = __sum;
18.      __tmps[thID] = Cs;
19.      __tmpe[thID] = Ce;
20.    }
21.    __sum = 0;
22.  }
23.
24.  void reductionCountTP(){
25.    int thID = omp_get_thread_num();
26.    sum += __sum;
27.    if(min > __min){
28.      min = __min;
29.      tmps =  __tmps[thID];
30.      tmpe = __tmpe[thID];
31.    }
32.  }
33.
34.   int main(int argc, char* argc[]){
35.     RV_Image* input_img;
36.     input_img = new RV_img();   //input an image
37.     input_img->reduction(reductionCountTP);
38.     input_img ->procBox(CountTP); //calling higher-order method
39.     return 0;
40.     }
```

Fig. 6. Template matching program after translation.

1: Some value is written to the global variable in a component function.

2: The global variable is not used in a component function which has data dependency between loop iterations.

If a global variable satisfies the both criteria, the previous preprocessor automatically inserts codes of reduction operations and translates the target variable into a reduction variable. When a global variable is detected in a component function and decided as a variable which needs a reduction operation, an additional reduction variable will be defined as a **thread-local storage (TLS)**. Thread-local variables can be specified by `__thread` directive and can be used for storing a local result which each thread calculates. After parallel processing has finished, values of reduction variables are gathered together.

However, the previous preprocessor cannot generate reduction codes for all the kinds of global variables because `__thread` directive cannot be used for the instances of RaVioli-specific classes. To solve this problem, we improve the preprocessor to be able to define a reduction variable for RaVioli-specific classes as an array which has as many elements as the number of threads. The number of threads is not specified by programmers but can be acquired by the preprocessor. RaVioli uses OpenMP[14] for parallel processing. OpenMP is suitable for parallelization of the image processing because OpenMP can parallelize loops easily by using compiler directives called *pragma*. By using the routine `omp_get_max_threads()` of OpenMP before translating program, the proposed preprocessor can acquire the maximum number of thread. As well, the proposed preprocessor also adopts the above criteria to decide whether global instances of RaVioli-specific classes need reduction operations or not.

Now, let us see how the proposed preprocessor automatically inserts codes of reduction operations and translates target global variables into reduction variables. Fig. 5 and Fig. 6 show template matching programs before and after the translation respectively. In both programs, the function `CountTP` is a component function for comparing the template-image and each partial-image which are cut out from the whole image. The function `Compare` called in the function `CountTP` is also a component function which calculates difference between two images. The variables `sum`, `min`, `tmps` and `tmpe` are global variables, hence the preprocessor translates these variables into reduction variables. In particular, the variables `tmps` and `tmpe` are the instances of RaVioli-specific classes.

Now, we explain how the proposed preprocessor translates a sequential RaVioli program into a parallel program. At first, the preprocessor declares thread-local variables `__min` and `__sum` as reduction variables for `min` and `sum` using `__thread` directive. (line 2 in Fig. 6). On the other hand, the variable `tmps` and `tmpe` are the instances of RaVioli-specific classes, hence reduction variables for `tmps` and `tmpe` are declared as arrays (line 4 in Fig. 6). Next, the preprocessor generates the code for calculating the local results (line 10 and line from 16 to 22 in Fig. 6) and the function for

gathering them (line from 24 to 32 in Fig. 6). The function for gathering the local results needs to be called by each thread at the end of parallel processing. In RaVioli, such a function can be processed exclusively by being passed to a higher-order method. Hence, the preprocessor inserts the call of this function into the program (line 37 in Fig. 6). As well, to get the thread ID for distinguishing storage area of each thread, the preprocessor inserts the call of the function `omp_get_thread_num` (line 14 and 25 in Fig. 6).

B. Load Balancing between Threads

As mentioned above, the proposed preprocessor automatically parallelizes a RaVioli program, and improves the processing precision. In this section, we explain a load balancing algorithm considering the sub-frames which have different precisions. When multiple threads run in parallel, the computation load of each thread should be balanced. Now, if each thread processes the same number of sub-frames, the computation load allocated to each thread may be unbalanced. This is because the processing load of each sub-frame varies depending on its stride value. Now, we explain an example of such an unbalanced situation using Fig. 7 and Fig. 8. Fig. 7 shows an example of an input frame. In Fig. 7, boxes indicate sub-frames and they are color-coded according to their stride values. When each thread processes the same number of sub-frames, the sub-frames are allocated to the threads as shown in Fig. 8. This task allocation can make the computation load of each thread imbalanced. Besides this task allocation, a dynamic scheduling is well known. This scheduling allocates tasks to each thread according to the processing time of the thread. However the cost of this scheduling is not small, and the load balancing of the computation load between threads is not optimazed. To prevent such load imbalance, we adopt a load balancing algorithm which considers the difference of processing load between sub-frames. For the content-aware resolution management, RaVioli sets suitable stride (e.g. base-stride, rough-stride and medium stride) to sub-frames according to their importance. Using these stride values, the ratio of the processing load of sub-frames can be calculated easily. Specifically, when the ratio of base, medium, and rough stride values is 1:2:4, the ratio of processing load can be estimated as 16:4:1. Using this estimated ratio, we aim to balance computation load of each thread.

Following the algorithm, each sub-frame is allocated to one of the threads in descending order of its processing load. The processing load of allocated sub-frames is accumulated to each threads. Then the thread which has the least total load will be selected and the next sub-frame will be allocated to the thread. With this algorithm, sub-frames are allocated to the threads as shown in Fig. 9 and more efficient parallel processing can be achieved.

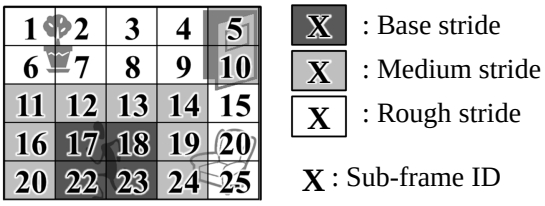


Fig. 7. An example of content-aware resolution mapping.

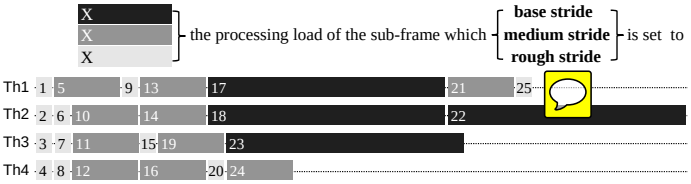


Fig. 8. An example of imbalanced task allocation.

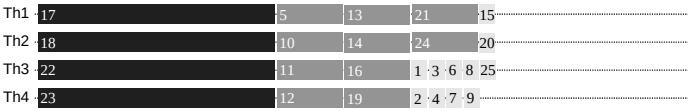


Fig. 9. Balanced task allocation considering the precision of each sub-frame.

V. Evaluation Results

We evaluated whether RaVioli with the proposed auto-parallelization mechanism can improve the processing precision than the previous RaVioli. The evaluation environment is shown in TABLE I.

For this evaluation, we used a template matching program for searching a face in profile. The input video stream is composed of 100 frames and the spatial input resolution is 1024×768. We set the ratio of base, midium, and rough stride values to 1:2:4. We evaluated and compared two video processing results; (a) without an auto-parallelization mechanism and (b) with the proposed auto-parallelization mechanism. Fig. 10 shows these resulting outputs of the 80th frame. In the result (a), the face in profile of the walking person can not be processed precisely. On the other hand in the result (b), deterioration in the precision of the face in profile of the walking person can be restrained than in the result (a). The above results lead to the conclusion that video processing applications with proposed RaVioli can achieve both of realtimeness and high precision processing results.

The fluctuation of spatial base stride is shown in Fig. 11. The horizontal axis shows the input frame indices, and the vertical axis shows the value of stride. The vertical gray line in the figure indicates the 80th frame shown in Fig. 10. In the result (b), spatial base stride can be kept lower than the previous RaVioli. On the other hand, in the result (b), temporal base stride can be kept lower than the previous RaVioli just like spatial base stride. TABLE II shows the average value of spatial stride and the average value of base stride. As shown in this table, deterioration in processing precision can be restrained by the proposed auto-parallelization mechanism.

(a) Resulting outputs without an auto-parallelization mechanism.

(b) Resulting outputs with the proposed auto-parallelization mechanism.

Fig. 10. Outputs of the 80th frames.

TABLE I
EVALUATION ENVIRONMENT.

OS	CentOS 6.4
CPU	Intel Core i7-4770 (4cores/8threads)
Frequency	4.0GHz
Memory	16GB
Compiler	gcc 4.6.0
Compiler options	-O3, -fopenmp

TABLE II
AVERAGE BASE STRIDES.

	spatial	temporal
without an auto-parallelization mechanism	3.71	2.75
with the proposed auto-parallelization mechanism	2.82	2.50

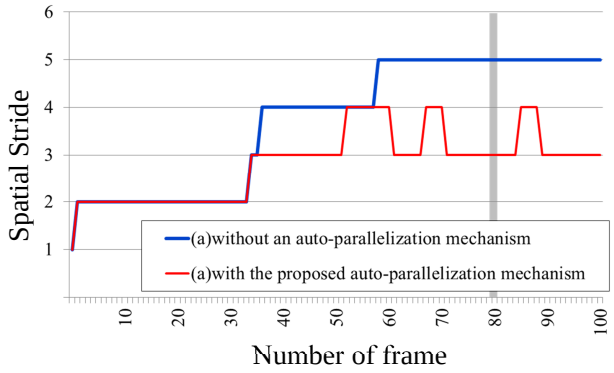


Fig. 11. The fluctuation of temporal base stride.

VI. CONCLUSION

In this paper, we improved the preprocessor which automatically translates a sequential RaVioli program into a parallel program. The preprocessor also provides a load balancing algorithm which is suitable for the content-aware resolution management.

Through an evaluation with a template matching program, it is found that the RaVioli program which is automatically generated by the proposed preprocessor can restrain deterioration in processing precision.

One of our future works is finding a better task allocation to threads for real-time video processing. We implemented a task allocation for balancing the computation load of each thread, however, for real-time processing, it is also important to shorten time which is taken to allocate tasks. If we consider this point, we can achieve further improvement of the output precision.

REFERENCES

[1] 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 2009*, vol. 1, Nov. 2009, pp. 321–329.

[2] T. Matsunaga, S. Ohira, T. Tsumura, and H. Matsuo, "Content-Aware Precision Control on a Real-Time Video Processing Library," in *Proc. 2013 Int'l Conf. on High Performance Computing and Simulation (HPCS2013)*, Jul. 2013, pp. 453–460.

[3] A. Garcia-Martin and J. M. Martinez, "Robust Real Time Moving People Detection in Surveillance Scenarios," in *Proc. 7th IEEE Int'l Conf. on Advanced Video and Signal Based Surveillance (AVSS'10)*. IEEE Computer Society, Aug. 2010, pp. 241–247.

[4] C. Kim, Y. Han, Y. Seo, and H. il Kang, "Statistical Pattern Based Real-time Smoke Detection Using DWT Energy," in *Proc. Int'l Conf.on Information Science and Applications*. IEEE Computer Society, Apr. 2011, pp. 1–7.

[5] K. Lin, J. Huang, J. Chen, and C. Zhou, "Real-time Eye Detection in Video Streams," in *Proc. 4th Int'l Conf. on Natural Computation*, vol. 06. IEEE Computer Society, Oct. 2008, pp. 193–197.

[6] J. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung, "Imprecise Computations," in *Proceedings of the IEEE*, vol. 82, Jan. 1994, pp. 83–94.

[7] H. Yoshimoto, N. Date, D. Arita, and R. Taniguchi, "Confidence-Driven Architecture for Real-time Vision Processing and Its Application to Efficient Vision-based Human Motion Sensing," in *Proc. 17th Int'l. Conf. on Pattern Recognition (ICPR'04)*, vol. 1, 2004, pp. 736–740.

[8] U. Köthe, "Generic programming for computer vision: The vigra computer vision library," http://hci.iwr.uni-heidelberg.de/vigra/, Sep. 2011.
[9] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision With the OpenCV Library*. O'Reilly & Associates Inc, 2008.
[10] 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.
[11] Greyc Laboratory, *Pandore: A library of image processing operators.*, 6.6.5 ed., Jul. 2014.
[12] 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*, pp. 98–103, 2007.
[13] 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.
[14] L. Dagum and R. Menon, "OpenMP: an Industry Standard API for Shared-Memory Programming," *IEEE Computational Science and Engineering*, vol. 5, pp. 46–55, 1998.