RAVIOLI: A PARALLEL VIDEO PROCESSING LIBRARY WITH AUTO RESOLUTION ADJUSTABILITY

Hiroko Sakurai[1], Masaomi Ohno[1], Shintaro Okada[2], Tomoaki Tsumura[1] and Hiroshi Matsuo[1]
[1] *Nagoya Institute of Technology* *Gokiso, Showa, Nagoya, Aichi, Japan*
[2] *Toyota Motor Corp.* *1, Toyota, Toyota, Aichi, Japan*

ABSTRACT

Video processing applications are now in demand on a great variety of platforms such as mobile devices or high performance servers. On the other hand, a great variety of performance is required for video processing applications such as high throughput, good accuracy, long battery life, and so on. Therefore, programmers today should study hard about various platforms and techniques under the pressure of necessity. This must make the burden heavy for them. This paper proposes a parallel video processing library RaVioli. RaVioli achieves self-optimizations for multi-core processors and self-adjustment of resolutions. RaVioli conceals two resolutions, frame rate and number of pixels, from users and provides dynamic and transparent resolution adjustability based on user-preferred priority parameters. This makes pseudo real-time video processing feasible for any platform by adjusting resolutions according to situations. Generally, video processing has some parallelism in its algorithm. For example, pixels in a frame have data parallelism, and many video processing algorithms can be divided into some processing stages which can be pipelined. Concealing resolutions makes implicit parallelism more obvious. Hence, RaVioli can parallelize programs semi-automatically.

KEYWORDS

Multi-core, real-time video processing, load adjustment, parallelization, pipelining

1. INTRODUCTION

It has been important to improve system/processor architecture and semiconductor efficiency for serving high performance to time-consuming applications. But now, multi/many-core processors like Niagara or Nehalem have come into wide use. It becomes, therefore, much important to improve compilers and libraries for multi-core processors being fully utilized. Especially, real-time video processing applications need much optimization. They should run on many platforms widely from mobile devices such as cell phones to high-end servers. To achieve best performance on each platform, programmers must have a considerable knowledge of processor/memory architectures of the target platform. Moreover, each platform has its own goal, and the goals are various such as long battery life, low temperature, extreme performance, and so on. Therefore, programmers must pay close attention to very diverse things irrespective of the video processing algorithms.

Meanwhile, video processing applications promise good performance on parallel systems such as multi-core processors. Pixels in video frames have data parallelism and image processing can be parallelized in multiple grains using such as SIMD instructions and block decomposition. Many of video processing programs can divided into multiple stages, and each frame can be processed in parallel by pipelining. Human detection, for example, consists of some stages; binarization, edge detection and Hough transformation.

However, parallelizing programs is not so simple. Rewriting programs is inevitable. For exploiting data parallelism, you must care about data dependencies and appoint reduction variables properly through the rewrite. For pipelining, programmers must care about the imbalance of computation loads between stages and make an effort to equalize them. These points of notice have made the burden heavy for programmers.

In this manuscript, we suggest a C++ library named **RaVioli** (Resolution Adaptable Video Operating Library) to address these problems. RaVioli automatically roughens the pixel rate and the frame rate when processing load comes high and real-time processing cannot be achieved. Whether the pixel rate or the frame rate should be roughened is decided by the *priority* specified by programmers. RaVioli has two parallel

processing functions for video processing. The one is semi-automatic block decomposition, and the other is automatic load balancing between pipeline stages.

2. RELATED WORKS

2.1 Video Processing Programming

For real-time video processing, adjusting the processing load is very important. Nevertheless, writing multiple routines with different algorithms has been the only solution for the load adjustment. One example is the imprecise computation model (ICM) (Liu, 1994; Yoshimoto, 2004). In this model, computation accuracy is varied corresponding to the given computation time. Confidence-driven architecture, which is based on the ICM, selects suitable routine dynamically and empirically from multiple routines written with different algorithms. Hence, programmers have to troublesomely implement multiple routines with different loads with this model.

VIGRA (Köthe, 2008) and OpenCV (Intel, 2001) are well-known video processing libraries. They aim at high-level descriptivity of video processing. Adopting template techniques similar to the C++ STL, VIGRA allows programmers to easily adapt given components to their programs. OpenCV provides many typical video processing algorithms as C functions or C++ methods. However, adjusting computation load is difficult to be implemented with these libraries.

The approach of a library RaVioli is completely different from these existing computation model and video processing libraries. 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 vary spatial/temporal resolution and can adjust processing load dynamically and automatically.

2.2 Parallel Programming

```
int i; int sum=0;
#pragma parallel reduction(+:sum)
for( i = 1; i <= 256; i++ )
       sum += 1;
```

Figure 1. Parallel program with OpenMP containing reduction operation

Intel Threading Building Blocks (TBB) (Reinders, 2007) is a C++ template library for multi-core processing. It conceals low-level threading activities. TBB manages task scheduling and programmers need not to take care of managing threads such as thread creation, synchronization and so on. All they need to do is mapping tasks to threads.

OpenMP (Dagum and Menon, 1998) is another solution for parallel programming. OpenMP is a standardized model of parallel programming for C/C++ and FORTRAN. With OpenMP, programmers should use compiler directives called *pragma* for specifying how to process code blocks. Fig.1 shows a summation program using OpenMP. The iterations in the *for* block below the parallel pragma will be processed in parallel. But in this case, the variable sum brings dependencies between iterations. Therefore, the summation process must go through following steps to get correct result.

– Makes thread-local variables for storing the sub-summation which each thread is responsible for.
– Each thread stores the result of sub-summation into its thread-local variable.
– Finally, gathers the local results and sums them all up.

To achieve these steps with OpenMP, programmers must specify additional reduction pragma as shown in Fig.1. TBB and OpenMP are good tools with sophisticated parallel programming model all right, but programmers must detect which block has parallelism and specify how to process it in parallel. On the other hand, RaVioli can parallelize programs semi-automatically by itself. In RaVioli programs, data dependencies and reduction requirements can be easily found. RaVioli automatically detects and manages them. Hence, programmers need not to pay attention to them.

3. OVERVIEW OF RAVIOLI

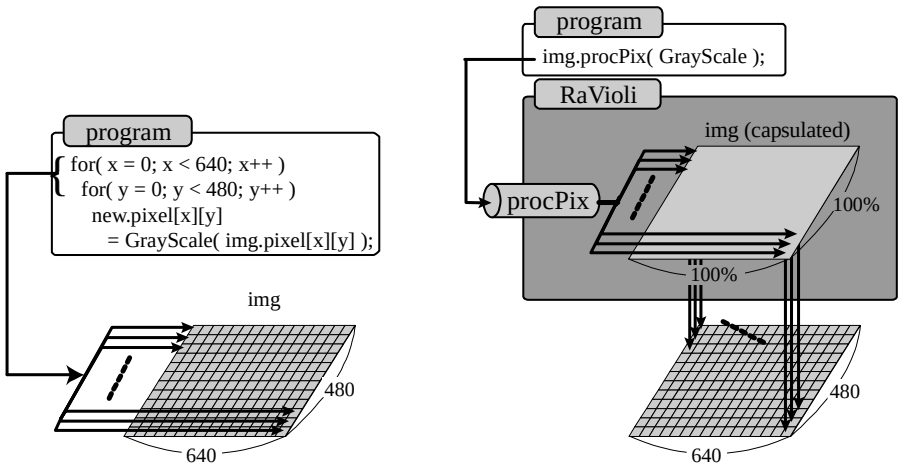
3.1 Abstraction of Video Processing

RaVioli proposes a new programming paradigm with which programmers can write video processing applications intuitively. RaVioli conceals **spatial resolution** (pixel rate) and **temporal resolution** (frame rate) of a video from programmers. We human beings naturally have no concept of resolutions through our visual recognition. For example, we can recognize motion objects in our view without any pixel or frame. However, pixels and frames are indispensable for motion object detection programs on computer systems.

Generally, motion object detection programs are implemented by using block matching algorithm, which searches the most similar block between current window and previous one. The similarity between image windows will be calculated by SAD (sum of absolute differences) or other alternative methods, and the methods should be implemented by cumulative pixel value differences. Resolutions are delivered from the requirement of quantitativeness on computers. Hence, programmers have to manage resolutions in their programs although resolutions are not required essentially for vision. Resolutions make programs unintuitive.

Generally, loop iterations are heavily used in video processing programs. When converting color to grayscale, for example, each pixel is converted in innermost iteration, and the process is repeated for every pixels by loops as shown in Fig.2(a). With RaVioli, this repeat for all pixels is done by RaVioli automatically, so programmers should only write a routine for one pixel as shown in Fig.2(b). GrayScale in Fig.2(b) is the routine defined by the programmer. What programmer should do are defining function which processes one pixel and passing the function to an image instance's public method procPix(). The proxPix() is defined as a higher-order function which applies a function passed as its argument to all pixels one after another. This framework allows programmers to be released from resolutions and the number of iterations.

Pseudo real-time processing and parallelization are also resolved by RaVioli. RaVioli conceals resolutions from programmers, therefore RaVioli can easily vary resolutions through real-time processing for load reduction. Moreover, the iteration unit is so distinct in RaVioli programs that the programs can be automatically data-parallelized. These functions are taken up in detail in following sections.



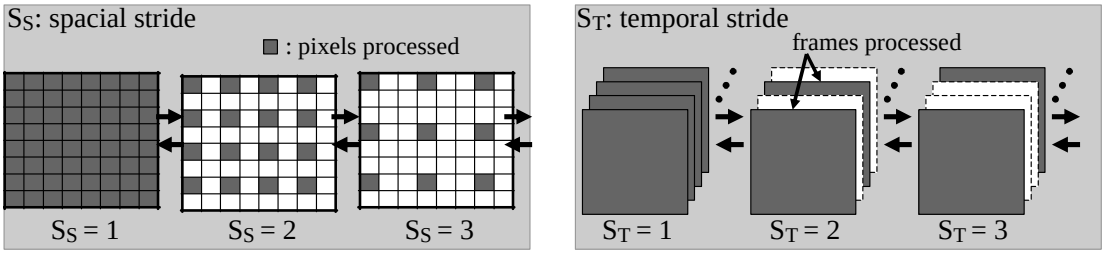
(a) Traditional image processing (b) Image processing with RaVioli

Figure 2. Digital image processing

3.2 Self-adjustment of Computation Load

RaVioli can vary video resolutions dynamically considering processing load. RaVioli periodically compares the frame capture interval and the processing time for one frame. When the processing time becomes larger than the capture interval, RaVioli considers it is overloaded and reduces resolutions. There are two resolutions: **spatial resolution** and **temporal resolution** in videos. Spatial resolution refers the number of

pixels contained in each frame. Temporal resolution refers the frame rate. RaVioli applies component functions to frames or pixels skipping on a certain **stride** in higher-order methods mentioned above. Roughening resolutions can be done by raising the stride value, and it leads to decreasing the computation load. Fig.3(a) shows which pixels are processed when special stride increases, and Fig.3(b) shows which frames are processed when temporal stride increases.



(a) Spatial resolution (b) Temporal resolution

Figure 3. Resolution changes

Priorities can be specified for telling RaVioli which resolution (special or temporal) should be kept. In a real-time video application, top priority will be given to temporal resolution, and RaVioli reduces spatial resolution. In other applications such as face authentication, top priority will be given to spatial resolution, and RaVioli reduces temporal one. What should be done for load adjustment is only specifying priorities.

The resolution priority is specified by a tuple of two values (P_S ,P_T) called **priority set**. P_S represents the priority of spatial resolution, and P_T the priority of temporal resolution. When (P_S ,P_T) = (3, 7) is specified, the priority ratio of P_S and P_T is recognized as 3:7, and RaVioli manages to keep spatial stride and temporal stride in the ratio of 7:3. When you want to port a video processing application to another platform, you should only change the values of priority set to a suitable values for the new platform without rewriting their program. Therefore you can easily implement a video processing application, which fulfills the performance demand and realizes real-time processing.

This algorithm for reducing resolutions is very simple and naive. However, this simplicity is very important. Many complement algorithms such as bi-linear, hyper-cubic, and so on are well known and they can be used. However, notice that the resolution-change function of RaVioli aims at reduction of calculations. Adding calculations for changing resolutions makes no sense. An application written with RaVioli can achieve real-time processing without any considerations. Sometimes the output will have low quality, but the application does not lose *realtimeness.* Moreover, defining priority set appropriately can control the inconvenience from the quality loss.

4. PARALLELIZATION IN RAVIOLI

4.1 Spatial Parallelization

A preprocessor is implemented which converts a sequential RaVioli program into a parallel program. With RaVioli, programmers first define a function which processes component parts of image such as a pixel or a small window. We call this function a **component function**. Then, programmers select a higher-order method and pass the component function to the method as shown in Fig.2(b). Higher-order methods are implemented by loops in RaVioli, and the component function will be applied over the entire image. The loop iterations have data parallelism, and they can be parallelized automatically inside RaVioli as Fig.4.

The preprocessor first traverses a RaVioli program searching higher-order method invocation and parallelize it with block decomposition using multi-threading. Through the parallelization, the preprocessor automatically decides whether a reduction operation is required or not. Hence, once a video processing program is written, multi-core parallel systems can be utilized without any rewrite.

When the preprocessor finds a parallelizable higher-order method invocation, it first ascertains whether the method invocation is *essentially sequential* or not. If it is not, the preprocessor parallelize the invocation, and generate reduction variables and reduction operations if required.

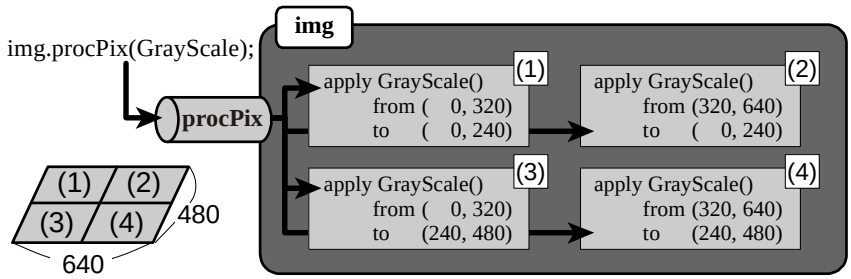


Figure 4. Spatial parallelization

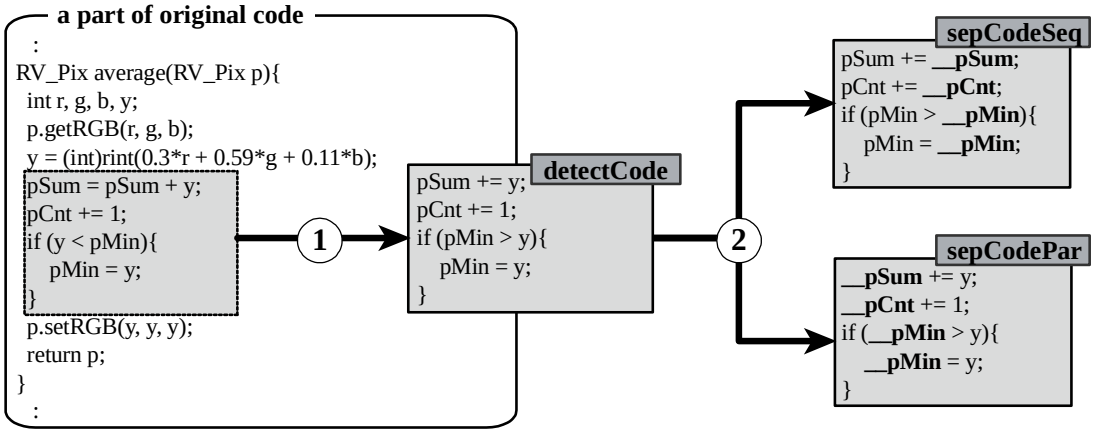


Figure 5. Transformation for a reduction

In an image processing program using RaVioli, some component functions are defined and they are applied to image instances through higher-order methods. Loop iterations that programmers used to write by themselves are now absorbed into RaVioli. Hence, any dependency between iterations appears as an assignment to a global variable in the component function. When one of the following cases[1] is found in component functions, the preprocessor decides that it is essentially sequential.

- An assignment expression to a global variable has no operator in the right-hand side. (e.g. foo = pixel.getR();)
- The right-hand side of an assignment expression to a global variable contains both + or - operator, and * or / operator. (e.g. foo = (foo + pixel.getR()) * 2;)

In short, if an assignment expression to a global variable in a component function satisfies both the associative law and the commutative law, the higher-order method invocation the component function is passed can be parallelized using additional reduction operation.

Now, let's see how our preprocessor generates reduction operations automatically. When a global variable is detected in a component function and decided as it 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 when you use Sun Studio C++ compiler or GNU C++ compiler. The reduction variable will be used for storing local results of block decomposed sub-processes. After all the sub-processes have finished, the values of the reduction variables will be gathered together.

Fig.5 shows translation of a component function which needs reduction operation through parallelization. The function average() is a simple component function which calculates the average of all pixel values in a image. pSum, pCnt and pMin are global variables. First, (1) preprocessor normalizes the code to make it easier to find reduction requirement. Next, (2) the code is divided into two codes shown as sepCodePar and sepCodeSeq in Fig.5. The former calculates the local averages and the latter gathers them. The variables __pSum, __pCnt, __pMin are reduction variables defined as thread local, and they corresponds to the global variables pSum, pCnt and pMin respectively. The sepCodePar code is executed in parallel by multiple threads, and after that, the sepCodeSeq code is executed once in master thread.

[1] There are more cases, but we omit them for want of space.

4.2 Assisting Pipeline Implementation

Many video processing applications can be accelerated with pipelining. RaVioli provides an interface to implement pipeline processing. For pipelining video processing, the whole process first has to be split into several *stages*. For example, a simple face detection program can be split into binarization, edge detection and Hough transform. Next, several threads should be created and assigned to the stages one by one. Furthermore, FIFOs are need to be expressly implemented and managed for data transfer between stages.

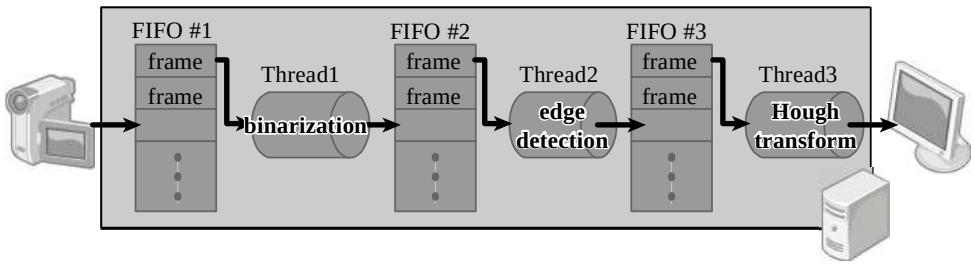


Figure 6. Pipelined video processing

```
RV_Pipedata* GrayScale(RV_Pipedata* data){
    //Gray-scale processing for a frame
    return data;
}
RV_Pipedata* Laplacian(RV_Pipedata* data){
    //Laplacian filter processing for a frame
    return data;
}
int main(){
    RV_Pipeline pipe;     //Making an instance of pipeline class
    pipe.setParam(7,3);   //Setting priority of resolutions
    pipe.push(GrayScale); //Creating a Gray-scale stage
    pipe.push(Laplacian); //Creating a Laplacian stage
    pipe.run();           //Starting pipelining
    return 0;
}
```

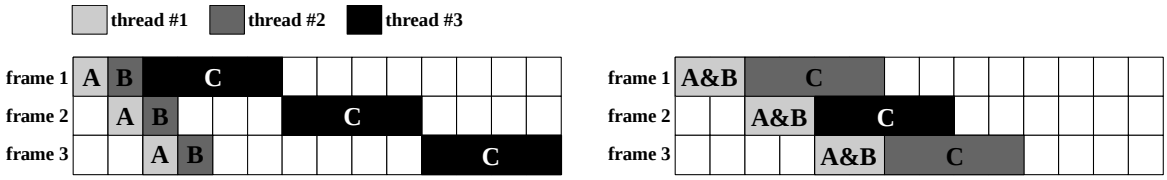
Figure 7. An example of pipelining program with RaVioli

An example of pipelined video processing is illustrated in Fig.6. Frames captured by the camera are stored in the FIFO (or pipeline register) #1 shown in Fig.6. The first stage *binarization* processes a frame and stores the result into the FIFO#2. Simultaneously, the stage *edge detection* picks up one data from FIFO#2, process it, and stores the result into #3. All stages can process sequential frames simultaneously. However, implementing pipeline is not simple. Processing loads differ each other between stages, and programmers must design stages carefully for balancing the loads. Furthermore, implementing additional units for pipelining such as FIFOs is not the essence of video processing and is troublesome for programmers.

RaVioli provides a pipelining interface against these nuisances. When pipeline stages are created through this interface, required pipeline units are automatically defined, and the load imbalance is automatically resolved. Programmers first define functions which should be assigned to the pipeline stages, and pass them one by one to a higher-order method `push` which a RaVioli pipeline instance has, as a sample code shown in Fig.7. The method `push` creates new thread, assigns the passed function to the thread as a new pipeline stage, prepares a FIFO for the stage, and connects them to the previously defined stage.

RaVioli keeps watch on the FIFOs. If there is no load imbalance between the stages, each FIFO will have only one frame at most in it. When RaVioli detects several overdue frames in some FIFO, RaVioli tries to integrate and parallelize stages automatically. Fig.8(a) shows an example of load imbalance. Three functions A(), B() and C() are assigned to pipeline stages, and the load of C is 4 times as large as A and B. The thread #3 assigned to C should stall for each frame, and throughput of whole pipeline will be limited by the bottle neck owing to C. RaVioli inquires the processing time for one frame of each stage, calculates the average of them, and tries to approximate the processing time of each thread to the average. If a pipeline stage is judged as overloaded, it will be parallelized. The parallelization is implemented by assigning multiple threads to one

pipeline stage. The multiple threads pick up frame data from the input FIFO by turns, and process them. On the other hand, multiple idle stages will be integrated into one stage if they are neighboring. A balanced pipeline by RaVioli is illustrated in Fig.8(b). Function A and B are assigned to one thread, and C is parallelized with two threads.



(a) w/o stage integration/parallelization (b) integrated/parallelized

Figure 8. Processing states of each frame

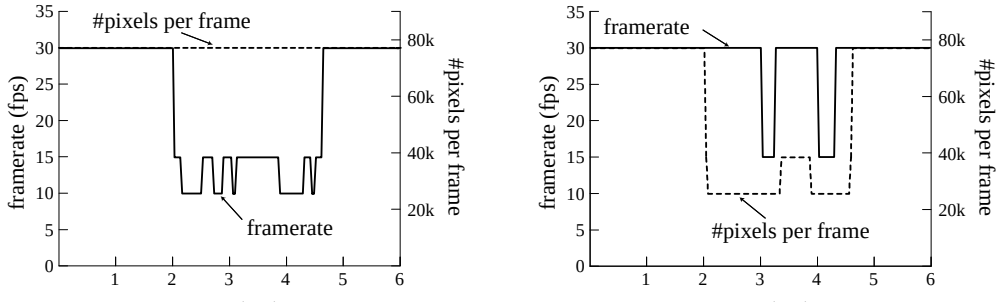
These two parallelization mechanisms can raise the throughput of RaVioli video processing programs. If these mechanisms are used effectively, the resolution fluctuations through real-time processing could be suppressed to the minimum.

5. EVALUATIONS

5.1 Real-time Processing

We have evaluated real-time processing performance with RaVioli through a frame difference program on AMD dual-core Opteron. This program calculates differences between adjoining frames so that motion objects are detected. The input video stream is 30fps and the spatial resolution is 320 x 240.

Evaluation is done with several priority parameters. Fig.9 shows the results with priority parameters (P_S, P_T) = (1, 0) and (P_S, P_T) = (7, 3). (P_S, P_T) represents a parameter set described in section 3.2. We observed resolution fluctuations for 6 seconds. From 2 to 4 second, the processor is burdened with another process and the amount of CPU resource which can be used by the evaluation program is reduced.



(a) (P_S, P_T) = (1,0) (b) (P_S, P_T) = (7,3)

Figure 9. Resolution change with several priorities

Fig.9(a) shows that temporal resolution is kept and only spatial resolution is degraded after 2 second. After the burden process has gone, around 4 second, the spatial resolution rises back to initial steady state. Fig.9(b) shows that both resolutions are degraded, and that the degradation of spatial resolution is larger than the one of temporal resolution. As above, we evaluated the resolution fluctuation mechanism of RaVioli for pseudo real-time processing. It is proved that RaVioli observes the fluctuation of usable CPU resource and varies the resolution appropriately. For examining the writeability, we have implemented various applications such as laplacian filter, template matching, Hough transform, image rotation, and so on, and have ascertained that they run correctly.

5.2 Pipelining Mechanism

The pipelining mechanism of RaVioli is evaluated on Sun UltraSPARC T1 (Leon, 2007). The sample program used for this evaluation has three stages and the stages are assigned to different threads. Here, assume that the load of third stage is 4 times as large as first and second stages. And note that temporal resolution is given first priority by priority set (P_S ,P_T) = (0, 1). Fig.10(a) shows an output image using pipelining without load balancing mechanism, and Fig.10(b) shows an output image with load balancing. We can see that the spatial stride is prevented from being roughened by the performance management with pipelining.

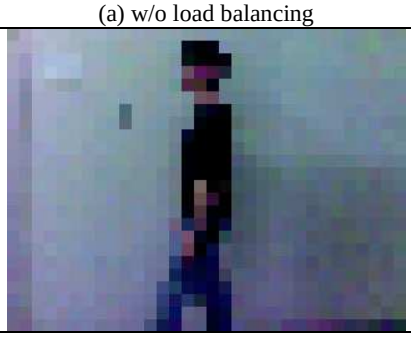
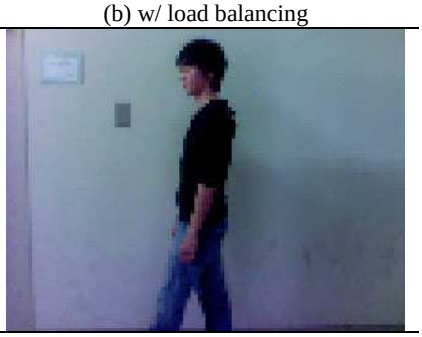
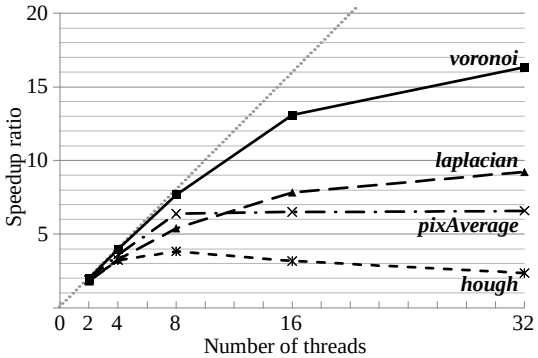
	(a) w/o load balancing	(b) w/ load balancing
Output image		
Spatial resolution	51 x 51 (stride = 11)	170 x 170 (stride = 4)

Figure 10. Output images w/o and w/ pipeline stage load balancing

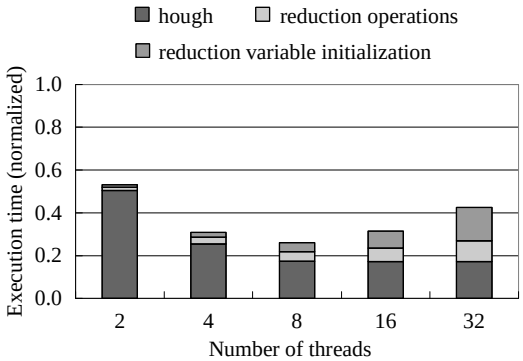
5.3 Parallelization with Block Decomposition

Spatial parallelization by semi-automatic block decomposition described in 4.1 is evaluated on Sun UltraSPARC T1 . Following programs are used for this evaluation. The two programs *pixAverage* and *hough* need reduction operation through parallelization.

- *voronoi*: A 2-dimensional discrete voronoi diagram is constructed by depositing different color per voronoi region. The regions are determined by calculating the nearest kernel point of each pixel.
- *laplacian*: An edge image is created by neighborhood processing.
- *pixAverage*: An average of all pixels in an image is calculated.
- *hough*: A voting curve in the $\rho-\theta$ parameter is figured for line detection.



(a) Speedup ratio



(b) Detailed execution time of hough

Figure 11. Sample programs executed on UltraSPARC T1

Fig.11(a) shows speedup ratio against sequential program. Sun UltraSPARC T1 has 8 processor cores, and is able to execute 32 threads simultaneously by the chip multithreading technology (CMT) (Spracklen and Abraham, 2005). The results of all programs show that the semi-automatic parallelization mechanism of RaVioli effectively utilizes the 8-core processor without any rewrite. The speedup ratios of all programs

increase monotonically not exceeding 8 threads. Two programs *voronoi* and *laplacian* continue to progress over 8 threads, and the speedup ratio exceeds 8 the number of cores.

However, over 8 threads, the performance of *hough* decreases. We itemized the total processing time of *hough* for detailed analysis. The result is shown in Fig.11(b). Each bar in Fig.11(b) is normalized to the execution time with only one thread. This result indicates that reduction overhead, which contains reduction operations and reduction variable initialization, grows expensive over 8 threads. This should be solved by improving preprocessor or RaVioli itself in the future work.

6. CONCLUSION

This paper proposes a video processing library RaVioli. RaVioli conceals the concept of *resolutions* and provides a novel programming paradigm. Hence video processing applications can be easily developed with RaVioli. RaVioli varies spatial/temporal resolutions dynamically and automatically for achieving pseudo real-time video processing. RaVioli can also parallelize video processing programs by semi-automatic block decomposition for each frame and by easy-to-use pipelining interface which can automatically balance the stage loads. For evaluations, we implemented several programs with RaVioli. The results also prove that the pseudo real-time processing mechanism works appropriately and that the parallelization mechanism contributes to good processing accuracy and good utilization of multi-cores.

Currently, power-saving techniques are going to be very important. In future work, we thus plan to implement an automatic power-saving function onto RaVioli. This function reduces resolutions automatically to cut down energy consumption when input frames have no/little changes. Memoization (Norvig, 1992; Tsumura, 2007) technique could be also used for power saving and good performance. RaVioli is implemented with Pthread library at the moment, but another important issue is making RaVioli adaptive to various platforms such as Cell/B.E. or CUDA GPUs. The evaluation results show the good writeability with RaVioli, but image reconstruction and frequency processing cannot be written with RaVioli at the moment. We should examine some new methods for them. Designing a new video programming language which cooperates with RaVioli is also left for future work.

ACKNOWLEDGEMENT

This research was partially supported by a Grant-in-Aid for Young Scientists (B), #21700028, 2009, from the Ministry of Education, Science, Sports and Culture of Japan.

REFERENCES

Dagum, L. and Menon, R. (1998) OpenMP: an Industry Standard API for Shared-Memory Programming. *In IEEE Computational Science & Engineering*, Vol.5, pp.46-55.

Intel Corp. (2001) *Open Source Computer Vision Library*.

Köthe, U. (2008) *VIGRA - Vision with Generic Algorithms*. 1.6.0 edn.

Leon, A.S. et al. (2007) A Power-Efficient High-Throughput 32-Thread SPARC Processor. *In IEEE Journal of Solid-State Circuits*, Vol.42, pp.7-16.

Liu, J. et al. (1994) Imprecise Computations. *In Proc. of the IEEE.* Vol.82, pp.83-94.

Norvig, P. (1992) *Paradigms of Artificial Intelligence Programming*. Morgan Kaufmann.

Reinders, J. (2007) *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly.

Spracklen, L. and Abraham, S.G. (2005) Chip Multithreading: Opportunities and Challenges. *In Proc. 11th Int'l Symp. on High-Performance Computer Architecture (HPCA-11).*, San Francisco, USA, pp.248-252.

Tsumura, T. et al. (2007) Design and Evaluation of an Auto-Memoization Processor. *In Proc. Parallel and Distributed Computing and Networks (PDCN2007)*, Innsbruck, Austria, pp.245-250.

Yoshimoto, H. et al. (2004) Confidence-Driven Architecture for Real-time Vision Processing and Its Application to Efficient Vision-based Human Motion Sensing. *Proc. 17th Int'l. Conf. on Pattern Recognition (ICPR'04),* Cambridge, UK, Vol. 1, pp.736-740.