RaVioli: a GPU Supported High-Level Pseudo Real-time Video Processing Library

Katsuhiko KONDO* Takafumi INABA* Hiroko SAKURAI†

Masaomi OHNO* Tomoaki TSUMURA* Hiroshi MATSUO*

* Nagoya Institute of Technology, Japan
camp@matlab.nitech.ac.jp

† OMRON Corp., Japan.

ABSTRACT

Real-time video processing applications such as intruder detection system are now in demand and being developed. However, on general purpose computers, it is difficult to guarantee that enough CPU resources can be surely be provided. We have proposed a pseudo real-time video processing library RaVioli for solving this problem. RaVioli conceals two types of resolutions, frame rate and the number of pixels, from programmers. This makes video and image processing programmings more intuitive, but the performance may be lower by the abstraction overhead. To solve this problem, this paper proposes an improvement of RaVioli for supporting GPU platforms. For using GPUs effectively, a deep knowledge about them has been required, and this would have been a burden to programmers. The proposition on this paper provides an easy-to-use framework for developers. They can benefit from GPU without rewriting their RaVioli programs and get high performance video processing. The experiment results with image/video processing programs show that the proposed method improves the performance about 151-fold/164-fold in maximum against traditional RaVioli without rewriting programs, and about 30-fold/4-fold in maximum against a native C++ program.

Keywords

real-time video processing, programming paradigm, video processing library, CUDA

1 INTRODUCTION

The demand of the systems, which highly requires real-time video processing, is rapidly increasing; such as intruder detection systems, automatic vehicle collision avoidance systems, and so on. 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. In spite of the advances, it is difficult to realize the real-time video processing on general-purpose operating systems, because it should be run by constant time interval. The main reason of the difficulty is the fluctuation in the throughput of frame rate 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) which guarantees pseudo real-time processing on general-purpose system platforms. RaVioli can regulate the throughput rate by automatically fluctuating spacial resolution and frame rate according to CPU usage and load. For such dynamical fluctuation of the resolutions, a programming fashion which is independent of the resolutions is required. RaVioli conceals two resolutions, the spacial resolution of an image and the frame rate of a video stream, from programmers for controlling the resolutions automatically at run-time. It makes possible to exclude the concept of resolutions, and developers can write video processing programs more intuitively.

However, RaVioli causes the decline of processing speed that comes from the abstraction overhead. Hence, to solve this issue, this paper proposes an improvement of RaVioli to support CUDA GPU platforms which are ideally suited to multimedia processing. The proposition of this paper is the method to provide an easy-to-use programming framework. Developers can implement real-time video processing programs without considering GPU architectures, and achieve high performance video processing.

2 RESEARCH BACKGROUNDS

2.1 Related Works

For real-time video processor, 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

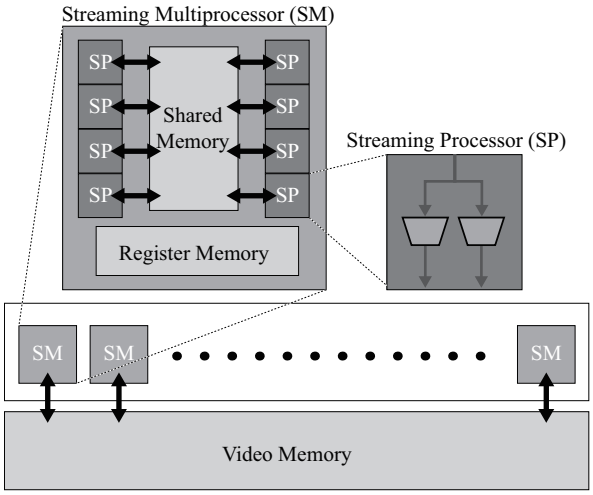


Figure 1: Brief architecture of GPU

that has been proposed is the imprecise computation model (ICM)[1, 2]. In this model, computation accuracy is varied corresponding to the given computation time. With the confidence-driven architecture, which is based on the 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.

VIGRA[3] and OpenCV[4, 5] 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[6] is completely different from these existing computation models or 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 spacial/temporal resolutions and can adjust processing load dynamically and automatically.

2.2 GPU and CUDA

A *GPU (Graphics Processing Unit)* is a specialized microprocessor for image/video processing. It has wide memory bandwidth and high processing performance. A brief architecture of a GPU shipped by NVIDIA Corp. is shown in Figure 1. Although GPUs are different according to their generations and families, they have a common architecture. There are dozens of *Streaming Multiprocessors (SM)* in one GPU, and for example in the GT200 series, each SM has eight *Streaming Processors (SP)*. The eight SPs in an SM can run in SIMD (Single Instruction Multiple Data) fashion. A GPU is a massively parallel multi-core processor, for example, a GT200 series GPU has 30 SMs, and consequently has 240 SPs.

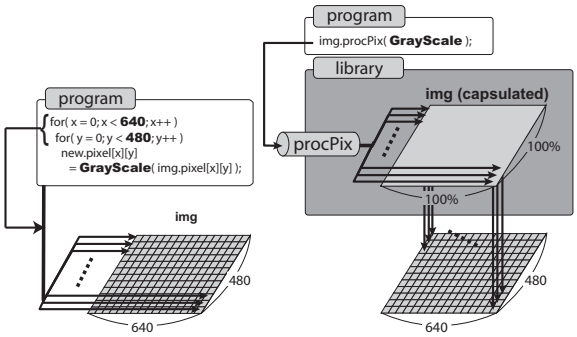
NVIDIA also provides *CUDA (Compute Unified Device Architecture)*[7] for GPU programming. CUDA is a parallel computing architecture for GPUs. CUDA also includes compilers and libraries, and provides APIs for GPU programming. Hence, developers can easily access memories of the computational units in GPUs using CUDA. GPUs can achieve high performance by executing massively parallel threads simultaneously. In the CUDA framework, a GPU can execute $65535 \times 65535 \times 512$ threads across all SPs. CUDA organizes these threads into two levels of units; *Grid* and *Block*. A Block is executed on an SM, and the threads in a Block can be identified by three-dimension indices (x, y, z). A set of Blocks is called Grid, and the Blocks in a Grid can be identified by two-dimension indices. How many threads are associated to a Block and how many Blocks per Grid are called as an *execution configuration*. Defining an appropriate execution configuration is a key for achieving good performance on GPU, but it is rather difficult for ordinary programmers.

Hence, some frameworks are proposed for CUDA programming. Baskaran et.al.[8] has proposed a translator for optimizing CUDA programs. It makes global memory accesses effective. The compiler framework optimizes affine loop nests based on a polyhedral compiler model. CUDA-lite[9] is another translator for CUDA programs. It generates a optimized code which uses appropriate GPU memories. Lee et.al.[10] has also proposed a optimization framework for GPU programs. However when using these frameworks, developers should pay attention to parallelism, and should add annotations or pragmas for getting efficient code. On the other hand, since RaVioli hides loop iterations from developers, essential parallelism or data dependencies between iterations are easily found automatically.

3 OVERVIEW OF RAVIOLI

3.1 Abstraction of Video Processing

RaVioli[6] 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 object motion in our view without any pixel or frame. However, pixels and frames are indispensable



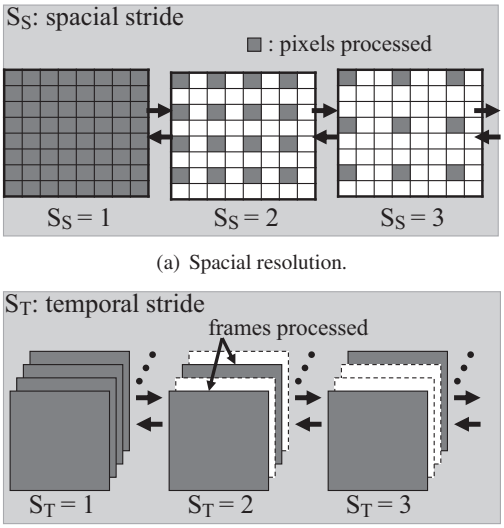
(a) Traditional program. (b) Program with RaVioli.

Figure 2: Digital image processing.

for motion object detection programs on computer systems.

For example, motion object detection programs are sometimes implemented by using a 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. In other words, the presence of resolutions makes programs unintuitive.

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 innermost iteration, and the process is repeated for every pixels by loop nests as shown in Figure 2(a). In RaVioli, an image is encapsulated in an RV_Image instance, and this repeating process for all pixels is done by RaVioli automatically, so programmers should only write a routine for one pixel as shown in Figure 2(b). GrayScale() in Figure 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 method 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. 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 images, videos are also encapsulated in RV_Video instances in RaVioli. Frames, the components of an RV_Video instance, are concealed from developers. An RV_Video instance also has several higher-order methods. Developers should only define a component function, which



(a) Spacial resolution.

(b) Temporal resolution.

Figure 3: Resolution changes.

manages one frame, and pass the function to an appropriate higher-order method for video processing.

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. Sakurai et.al. have taken these functions up in detail in [6].

3.2 Self-Adjustment of Computation Load

RaVioli can dynamically vary video resolutions considering processing load. RaVioli periodically compares the frame capturing 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, and 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. Figure 3(a) shows which pixels are processed when special stride increases, and Figure 3(b) shows which frames are processed when temporal stride increases.

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 authentica-

tion, 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 a *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. Therefore a video processing application, which fulfills the performance demand and realizes real-time processing, can be easily implemented.

This algorithm for reducing resolutions is very simple and naive. However, this simplicity is very important. Many complement algorithms such as bi-linear, hypercubic, and so on are well known and they can be used. However, notice that the function of changing resolutions of RaVioli aims at reducing 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 CUDA SUPPORT FOR RAVIOLI

In this section, CUDA-supported RaVioli (RaVioli/CUDA) and a translator which converts traditional RaVioli programs to programs for RaVioli/CUDA are proposed.

4.1 Execution Model of Image Processing with CUDA

Developers can use several memories of GPU with CUDA. Each memory has different access speed and size. For achieving high performance image processing, developers should use as fast memory as possible. For using fast memories, data should be transferred from main memory to GPU memories. Considering these memories and execution configurations needs deep knowledge and dexterity.

In this paper, we propose an extension of higher-order methods of RaVioli which supports CUDA API. Invoking these higher-order methods, developers can use GPUs without considering GPU memories, execution configurations and other troublesome steps. Figure 4 briefly shows how a GrayScale() function will be applied to an image by invoking the extended higher-order method cudaProcPix(). GrayScale() which is to be passed to cudaProcPix() should be defined as a *kernel function*. In CUDA, a kernel function specifies the code to be executed by all threads in parallel.[7]

When cudaProcPix() is invoked with a component function GrayScale(), RaVioli/CUDA allocates GPU

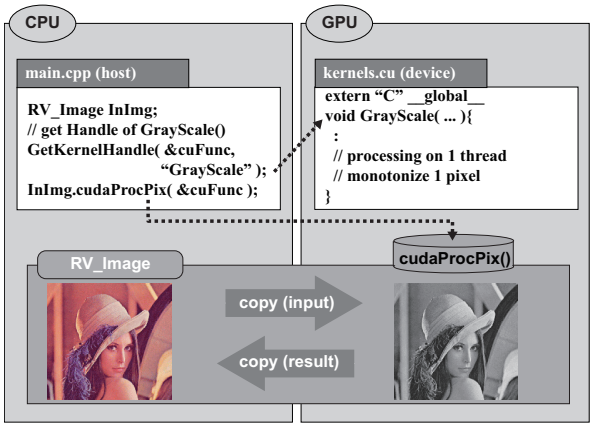


Figure 4: Brief execution model of RaVioli/CUDA

memories and transfers image data from main memory to GPU memories. After that, an execution configuration is automatically defined according to the input image, and GrayScale() is applied to the whole input image. When completing the application, RaVioli/CUDA transfers the result to the main memory on CPU and deallocates GPU memories. For each other higher-order method of RaVioli, an associated CUDA-supported method is defined.

4.2 Execution Model of Video Processing with CUDA

As same as RV_Image class, CUDA-supported higher-order methods for RV_Video class are also defined. The methods for RV_Video not only conceals data transfer between CPU and GPU, but also parallelize the data transfer and kernel function execution automatically by using *CUDA stream*.

In CUDA, the execution of a kernel function and the data transfer between host (CPU) and device (GPU) can be overlapped by using multiple CUDA streams. A CUDA stream is defined as a sequence of CUDA operations which are executed in-order. Multiple CUDA streams can be declared and used simultaneously. Each data transfer between host and device and each execution of kernel function can be assigned to one of the defined CUDA streams. A host-device data transfer and a kernel function execution on different CUDA streams can be executed in parallel.

In RaVioli/CUDA, two CUDA streams are automatically declared when an RV_Video is instantiated. When a higher-order method of the RV_Video instance is invoked, each frame of the video is assigned to the two CUDA streams alternately. The execution model is illustrated in Figure 5.

First, the stream #1 transfer the frame #1 from host to device. When the transfer completes, the stream #2 can transfer the frame #2, and the stream #1 applies kernel

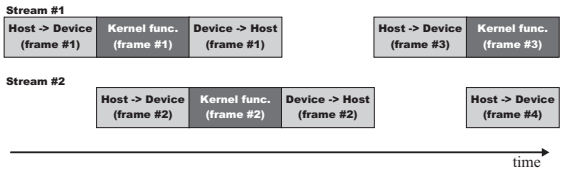


Figure 5: Pipelining with CUDA streams.

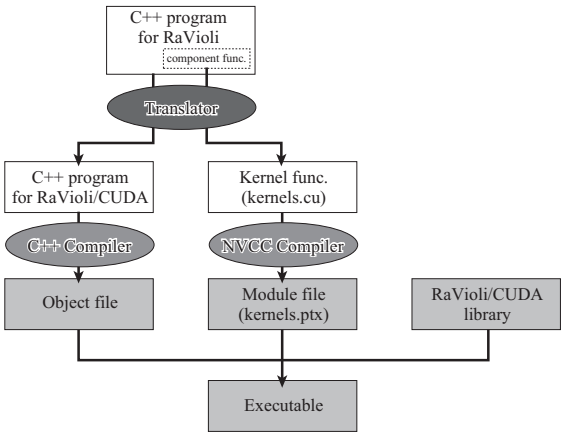


Figure 6: Compilation flow with translator.

function to the frame #1 simultaneously. Notice that the stream #2 cannot transfer the frame #3 as soon as the result of the frame #1 is sent back to the host, because the stream #2 will be send the result of the frame #2. This brings a pipeline bubble. However, in the ideal case in which every stage takes same latency, the throughput raises 1.5-fold.

4.3 Translator and Code Conversion

As described above, RaVioli/CUDA can provide an easy-to-use CUDA programming framework for developers. Higher-order methods of RaVioli/CUDA conceal almost all steps for using CUDA such as device handling, memory allocation, execution configuration, and so on from developers. However, developers still should modify their traditional RaVioli programs. For example in Figure 4, the developer should get a handler for a kernel function by calling GetKernelHandler(), and rewrite the higher-order method procPix() to the associated CUDA-supported higher-order function cudaProcPix(). The component functions also should be rewritten to kernel functions for being adapted to CUDA-supported higher order methods. To dissolve these troubles, a translator which converts traditional RaVioli programs to RaVioli/CUDA programs is also proposed in this paper. Figure 6 shows a compilation flow with the translator.

Figure 7 shows an example program which converts color images to grayscale. The translator converts such programs to two program files; `main.cpp` for host CPU and `kernels.cu` for GPU device. These program files are compiled by C++ compiler and CUDA compiler `nvcc`, and assembled to an executable. The result of conversion is shown in Figure 8 and Figure 9.

```
1  int main(int argc, char* argv[]){
2  RV_Image image;
3      :
4    image.procPix( GrayScale );
5      :
6  }
7
8  void GrayScale(RV_Pixel p1){
9    int ave = ( (p1.getR()) + p1.getG() + p1.getB() ) / 3;
10   p1.setRGB( ave, ave, ave );
11 }
```

Figure 7: A simple grayscale program with traditional RaVioli.

```
1  /* main.cpp */
2  RV_CudaDevice device;
3  int main(int argc, char* argv[]){
4    RV_Image image;
5    :
6    device.RaCudaInit(); /* initialize device */
7    CUfunction cuFunction;
8    device.GetKernelHundle( &cuFunction, "GrayScale" );
9    image.cudaProcPix( &cuFunction );
10   :
11   device.RaCudaExit(); /* finalize device */
12 }
```

Figure 8: Main program translated from Figure 7.

```
1  /* kernels.cu */
2  extern "C" __global__ void
3  GrayScale( RV_Pixel* idata, RV_Pixel* odata, int width, int height ){
4    int x = blockDim.x * blockIdx.x + threadIdx.x;
5    int y = blockDim.y * blockIdx.y + threadIdx.y;
6    RV_Pixel p1;
7    if( x < width && y < height ){
8      p1 = idata[ y * width + x ];
9      int ave = ( p1.getR() + p1.getG() + p1.getB() ) / 3;
10     odata[y * wid + x].setRGB( ave, ave, ave );
11   }
12 }
```

Figure 9: Kernel program translated from Figure 7.

In the main program, the invocation of procPix() in Figure 7 is converted to cudaProcPix() in Figure 8. A statement of GetKernelHandler() is added in main() for getting a kernel handler for the component function GrayScale(). RaCudaInit() and RaCudaExit() are functions provided by RaVioli/CUDA for CUDA device initialization and finalization respectively.

On the other hand in the kernel program, the component function GrayScale() is converted to a kernel function. A kernel function expresses a process for one thread. In Figure 9, the kernel function GrayScale() is defined as it processes one pixel on one thread. The definition of GrayScale() also makes continuous threads to process continuous pixels by calculating indices. This

is for coalesced access of CUDA memories. Memory accesses to global memory by continuous sixteen threads in each *Block* can be issued in parallel by this code conversion.

The translator searches higher-order method invocations through RaVioli programs, and generates associated code for RaVioli/CUDA with converting component functions to kernel functions. In this simple example program, there need no reduction operation for parallelization. However, in RaVioli programs, whether reduction operations are required or not can be easily detected, because any dependency between iterations appears as an assignment to a *global variable* in the component function.[6]

Enumerating translation rules in detail is left out for want of space. There are additional functions of the translator as follows.

Optimizing Data Transfers Many of video processing programs consist of multiple stages, and the stages can be pipelined. Since transferring data between CPU and GPU on each stage of the pipeline is redundant, the translator optimizes these data transfers. In the output program converted by the translator, the data are transferred from CPU to GPU only once at the first stage (i.e. the first invocation of a higher-order method), and the result is transferred from GPU to CPU only once at the last stage.

Using Page-Locked Memory With CUDA, two types of CPU host memory are available. The one is heap memory, and the other is page-locked host memory. Page-locked host memory is mapped into the address space of the GPU device, and can be accessed directly. Moreover, data copy between page-locked host memory and GPU can be fast and asynchronous. The translator converts programs for using page-locked memory automatically.

5 EVALUATION RESULTS

A CUDA extension for RaVioli and a translator described in section 4 were implemented, and evaluated with several image/video processing programs. The evaluation environment is shown in Table 1.

5.1 Evaluation of Image Processing

We used three programs which are grayscale, emboss filter and template matching for evaluating image processing. The evaluation results are shown in Table 2. In Table 2, *Baseline* denotes a program written in native C++, *RaVioli* denotes a program with traditional RaVioli and *RaVioli/CUDA* denotes a program with CUDA-supported RaVioli described in this manuscript. The size of the image which was used for grayscale and emboss filter was 512×512 pixels. For template matching, the base image has 395×372 pixels and the template image has 70×72 pixels.

OS	Fedora9
CPU	Core2Quad
Frequency	2.83GHz
Memory	3GB
GPU	GeForce GTX280
Number of multiprocessors	30
Number of cores (SP)	240
CUDA version	2.2 (Driver API)
Compute capability	1.3
Compiler	gcc
Compile options	-O3

Table 1: Evaluation environment.

Workloads	*Baseline*	*RaVioli*	*RaVioli/ CUDA*
Grayscale	0.83	2.89	1.21
Emboss filter	2.08	18.62	1.30
Templ. matching	1902.45	9512.69	62.62

Table 2: Execution time. (ms)

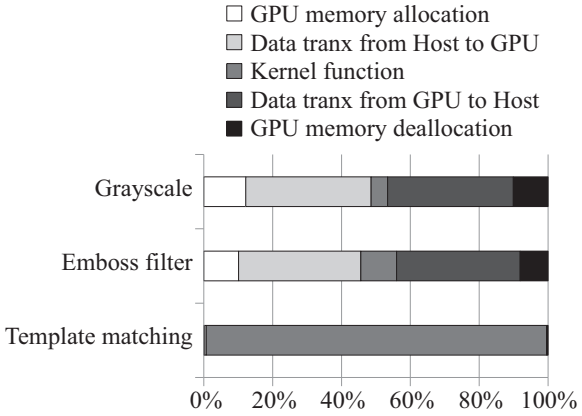


Figure 10: Breakdown of processing time with RaVioli/CUDA

As we can see in Table 2, *RaVioli/CUDA* achieves performance gains of 2.3-fold, 14.2-fold and 151.8-fold on grayscale, emboss filter and template matching respectively, against traditional RaVioli without rewriting programs. Typically on template matching, *RaVioli/CUDA* also achieves about 30-fold speedup against *Baseline*.

Nevertheless, the performance of RaVioli/CUDA on grayscale is still inferior to *Baseline*. Hence, the breakdown of processing time with *RaVioli/CUDA* was also evaluated. The result is shown in Figure 10. As we can see in Figure 10, the kernel function execution dominates the whole execution time on the template matching program, and the parallelization on GPU brings a

good result. On the other hand, the kernel function execution accounts for about only 4% of total execution on the grayscale program, and the data transfer overheads between host CPU and GPU device are dominant. This should prevent performance gain on the grayscale program. However, a program with small kernel function does not bring large latency and will not cause a problem on real-time video processing essentially.

5.2 Evaluation of Video Processing

The performance of video processing with RaVioli/CUDA was also evaluated. We have evaluated four models with an edge detection program. Roughly speaking, the edge detection processing consists of three stages; converts input frames to grayscale, binarize it, and detects object edges. The results of processing time for ten frames are shown in Figure 11.

RaVioli/CUDA achieved about 90-fold speedup against traditional RaVioli without rewriting program, and over 2-fold speedup against the baseline program. Furthermore, RaVioli/CUDA with CUDA stream achieved 164-fold speedup against traditional RaVioli, and about 4-fold speedup against the baseline. As we can see in the breakdown of the third bar, the latency of data transfer is longer than the latency of kernel function execution. However, the result overcomes the ideal 1.5-fold speedup mentioned in Chap. 4.2. This can be explained by additional functions of the translator mentioned in Chap. 4.3.

As a result, RaVioli/CUDA provides a high-level programming framework for developers. Developers can use GPU without any knowledge and consideration, and can achieve high performance by using RaVioli/CUDA. On expensive programs, GPU abilities can be easily brought out by RaVioli/CUDA, and on lightweight programs, RaVioli/CUDA can limit the abstraction overhead of RaVioli effectively.

6 CONCLUSIONS

In this paper, we have proposed an improvement of RaVioli for supporting CUDA GPU platforms. RaVioli is a pseudo real-time video processing library which conceals spacial/temporal resolutions from programmers and changes resolutions automatically for adapting to currently available CPU resource. RaVioli/CUDA not only allows developers to be free from considering GPU architectures, but also easily brings out the performance in GPU devices.

The evaluations with several image and video processing programs have been conducted. The results with image processing programs have shown that RaVioli/CUDA achieves 151-fold speedup in maximum against traditional RaVioli without rewriting programs, and also achieves about 30-fold speedup against native C++ programs. The results with a video processing program of edge detection has shown

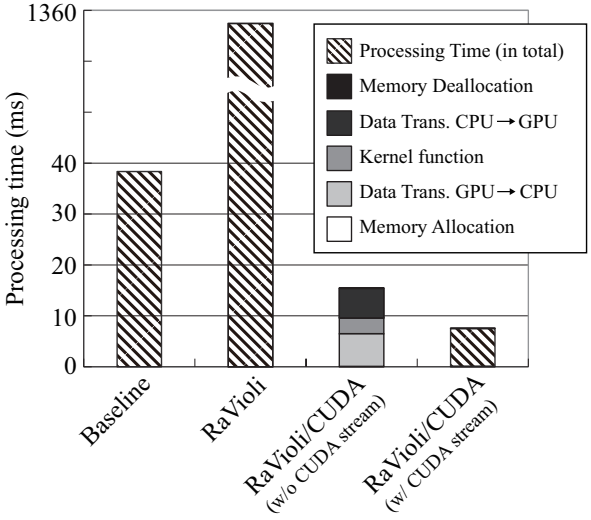


Figure 11: Evaluation of Video Processing.

that RaVioi/CUDA achieves about 164-fold speedup against traditional RaVioli and about 4-fold speedup against a native C++ program.

Possible improvement of this study is modifying execution configuration appropriately and dynamically. This can make kernel functions to run more effectively. Now, RaVioli has good writeability, and many programs such as edge detection, circle detection, hough transform, and so on can be written with RaVioli. However, image reconstruction and frequency processing are hard to be written with RaVioli at the moment. We should examine some new higher-order methods for them. Designing a new video programming language which cooperates with RaVioli is also left for our future work.

ACKNOWLEDGEMENTS

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

[1] J.W.S. Liu, Wei-Kuan Shih, Kwei-Jay Lin, R. Bettati, and Jen-Yao Chung. Imprecise Computations. In *Proceedings of the IEEE*, volume 82, pages 83–94, Jan. 1994.

[2] Hiromasa Yoshimoto, Naoto Date, Daisaku Arita, and Rinichiro Taniguchi. Confidence-Driven Architecture for Real-time Vision Processing and Its Application to Efficient Vision-based Human Motion Sensing. In *Proc. of the 17th Int'l. Conf. on Pattern Recognition (ICPR'04)*, volume 1, pages 736–740, 2004.

[3] Ullrich Köthe. *VIGRA - Vision with Generic Algorithms*, 1.6.0 edition, Aug. 2008.

[4] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer Vision With the OpenCV Library*. O'Reilly & Associates Inc, 2008.

[5] Intel Corp. *Open Source Computer Vision Library*, 2001.

[6] Hiroko Sakurai, Masaomi Ohno, Tomoaki Tsumura, and Hiroshi Matsuo. RaVioli: a Parallel Video Processing Library with Auto Resolution Adjustability. In *Proc. IADIS Int'l. Conf. Applied Computing 2009*, volume 1, pages 321–329, Nov. 2009.

[7] NVIDIA Corp. *NVIDIA CUDA Programming Guide*, 2.0 edition, Jun. 2008.

[8] Muthu Manikandan Baskaran, Uday Bondhugula, Sriam Krishnamoorthy, Atanas Rountev, and P.Sadayappan. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In *ICS'08: Proc. of 22nd Annual Intl. Conf. on Supercomputing*, pages 225–234. ACM, 2008.

[9] Sain-Zee Ueng, Melvin Lathara, Sara Baghsorkhi, and Wen mei Hwu. CUDA-lite: Reducing GPU Programming Complexity. In *Proc. of 21st Annual Workshop on Languages and Compilers for Parallel Computing (LCPC 2008)*, pages 1–15, 2008.

[10] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. In *Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 44, pages 101–110. ACM, 2009.

APPENDIX

Another example of code conversion by the translator is shown in this appendix. Figure 12 is a template matching program written with traditional RaVioli. The codes shown in Figure 13, Figure 14 and Figure 15 are generated by the translator from the code in Figure 12.

```
1  RV_Image tp_image;
2  RV_Coord start;
3  RV_Coord end;
4  int sad;
5
6  void SAD(RV_Pixel p1, RV_Pixel p2){
7    int abs = p1.absDiff(p2);
8    sad += abs;
9  }
10
11 void TPmatching(RV_Image imageSmall,
12                   RV_Coord startNow, RV_Coord endNow){
13   sad = 0;
14   int min = INT_MAX;
15   imageSmall.procImgComp(SAD, tp_image);
16   if(min > sad){
17     min = sad;
18     start = startNow;
19     end = endNow;
20   }
21 }
22
23 int main(int argc,char* argv[]){
24   RV_Image* input_image = new RV_Image(argv[1]);
25   RV_Image* output_image = new RV_Image(argv[1]);
26   tp_image = new RV_Image(argv[2]);
27   input_image->procBox(TPmatching,
28                        input_tp->getStartCoord(),
29                        input_tp->getEndCoord());
30   output_image->writeRect(start,end);
31   return 0;
32 }
```

Figure 12: A template matching program with traditional RaVioli.

The component function SAD() is converted to the SAD() in Figure 14, and the component function TP-

```
1  /* main.cpp */
2
3  #include "ravioli.h"
4  #include "cutil.h"
5
6  RV_Cuda device;
7
8  CUtexref cuTexTPref; // texture reference for template image
9  CUarray d_TPimage;
10 int3 result;
11
12 void TPmatching(RV_Image* image){
13   CUfunction cuFunction;
14   CUfunction cuFunction2;
15   device.GetKernelHundle(&cuFunction, "TPmatching_kernel");
16   device.GetKernelHundle(&cuFunction2, "reduction_kernel");
17   cuParamSetTexRef(cuFunction,
18                    CU_PARAM_TR_DEFAULT,
19                    cuTexTPref);
20   result = image->cudaProcBox(&cuFunction,
21                               tp_image->Width,
22                               tp_image->Height,
23                               &cuFunction2);
24 }
25
26 int main(int argc, char* argv[]){
27   RV_Image* input_image = new RV_Image(argv[1]);
28   RV_Image* tp_image = new RV_Image(argv[2]);
29
30   device.RaCudaInit();
31   device.GetTexrefHundle(&cuTexTPref,"texTP");
32
33   tp_image->TexRefSetImage(&d_TPimage, &cuTexTPref);
34   TPmatching(input_image);
35   cutilDrvSafeCall(cuArrayDestroy(d_TPimage));
36   image->writerect(result.x,result.y);
37
38   device.RaCudaExit();
39   return 0;
40 }
```

Figure 13: Main program translated from Figure 12.

matching() is Figure 13 and TPmatching_kernel() in Figure 13.

First, the translator finds the invocation of procBox() at line 27 in Figure 12, and tries to translate the component function TPmatching(). The procBox() is one of the higher-order methods of RV_Image instance, and it is for appling a component function repeatedly inside a certain box defined by two coord arguments.

In the main program shown in Figure 13, TPmatching() is defined. It gets kernel hundlers for kernel functions, sets up texture reference, and passes kernel functions to the higher-order method cudaProcBox(), which is the CUDA-supported version of proxBox().

TPmatching() in Figure 13 is only a wrapper function, and the essence of TPmatching() is translated to TPmatching_kernel() in Figure 14. It calculates sum of absolute differences by calling the function SAD(). Now, SAD() is called from device code. Hence, __device__ qualifier is added to SAD().

Thread-local results are stored in the data4reduction[] array. In Figure 12, the variable *sad* is defined as a

```
/* kernel.cu (module) */

texture<int, 2, cudaReadModeElementType> texTP;
__device__ int SAD(int* idata, int wid, int hei,
                   int widBox, int heiBox, int x, int y){
  int sad = 0;
  int p1, p2;
  for(int j = 0; j < heiBox; j++){
    for(int i = 0; i < widBox; i++){
      p1 = idata[ (y + j) * w + (x + i) ];
      p2 = tex2D(texTP, i, j);
      int abs = absDiff(p1, p2);
      sad += abs;
    }
  }
  return sad;
}

extern "C"
__global__ void
TPmatching_kernel(int* idata, int4* data4reduction,
                  int wid, int hei, int widBox, int heiBox){
  int x = blockDim.x * blockIdx.x + threadIdx.x;
  int y = blockDim.y * blockIdx.y + threadIdx.y;
  int incX = gridDim.x * blockDim.x;
  int incY = gridDim.y * blockDim.y;
  int sad;
  int min = INT_MAX;
  for(int j = y; j < (hei - heiTP); j += incY){
    for(int i = x; i < (wid - widBox); i += incX){
      sad = SAD(idata, wid, hei, widBox, heiTP, i, j);
      if(sad < min){
        data4reduction[y * 256 + x].z = sad;
        data4reduction[y * 256 + x].x = i;
        data4reduction[y * 256 + x].y = j;
      }
    }
  }
}
```

Figure 14: Kernel module program translated from Figure 12.

global variable and overwritten in the component function TPmatching(). This lets the translator know that there needs a reduction operation for the variable *sad*. Hence, the code for reduction shown in Figure 15 is also generated.

The code in Figure 15 reduces the thread-local results. Gathering the data over threads on shared memory in each *Block*, the minimum value and its coordination is settled, and the process is repeated over all *Blocks* by *for* loop.

The code through the line 19 to 29 in Figure 15, sixteen threads in each *Block* access continuous addresses in shared memory. Hence, bank conflict and Warp divergence can be avoided.

```
/* reduction code */

extern "C"
__global__ void

reduction_kernel(int4* data4reduction, int4* g_odata){
  __shared__ int sdatax[256];
  __shared__ int sdatay[256];
  __shared__ int sdataz[256];

  // from Global Memory to Shared Memory
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
  sdatax[tid] = data4reduction[i].x;
  sdatay[tid] = data4reduction[i].y;
  sdataz[tid] = data4reduction[i].z;
  __syncthreads();

  // reduction operations on Shared Memory
  for(unsigned int s = blockDim.x / 2; s > 0;s >>= 1){
    if(tid < s){
      if(sdataz[tid] > sdataz[tid + s]){
        sdatax[tid] = sdatax[tid + s];
        sdatay[tid] = sdatay[tid + s];
        sdataz[tid] = sdataz[tid + s];
      }
    }
    __syncthreads();
  }

  if(tid == 0){
    g_odata[blockIdx.x].x = sdatax[0];
    g_odata[blockIdx.x].y = sdatay[0];
    g_odata[blockIdx.x].z = sdataz[0];
  }
}
```

Figure 15: Reduction operations generated from Figure 12.