This paper is author's private version of the paper published as follows:
Proc. of 7th Int'l Conf. on Signal-Image Technology and Internet-Based Systems (SITIS2011), pp.245–252
Copyright (C) 2011 IEEE
DOI: 10.1109/SITIS.2011.66

A GPU-suppored High-Level Programming Language for Image Processing

Ami ONO*, Katsuhiko KONDO*, Takafumi INABA*, Tomoaki TSUMURA* and Hiroshi MATSUO*

*Nagoya Institute of Technology
Gokiso, Showa, Nagoya, Japan
Email: camp@matlab.nitech.ac.jp

***Abstract*—Real-time image/video processing applications are now in demand with the advance of general purpose computers and mobile devices. However, programmers have to handle the digital images, and be aware of the resolutions and pixels. This makes image processing programming unintuitive. On the other hand, image/video processing typically has data parallelisms, and the performance gains are expected on GPUs. CUDA is developed for GPUs but writing the image/video processing programs efficiently with CUDA needs many CUDA-specific operations. They are not the essence of image/video processing and bother programmers. We have proposed a high-level video processing library RaVioli for solving this problem. RaVioli allows programmers to be unaware of resolutions, but there are some restrictions for the programming. Hence, this paper proposes a more intuitive programming language for image/video processing and a translator for the language. By using the translator, programmers can benefit from GPUs without the knowledge about both the GPU architecture and the CUDA APIs, and achieve performance gains.**

I. Introduction

Video processing applications are now in demand on a great variety of platforms such as mobile devices or high performance servers. As mobile devices have been developed drastically, it is especially important to achieve high performance image/video processing on embedded devices. Generally, the performance of the processor for these embedded devices might be lower than that of general-purpose processors, and the power consumption should be kept lower from the viewpoint of the battery life. Therefore, it is essential for developers to handle the limited resources efficiently in order to realize real-time video processing on these devices. Furthermore, for implementing image/video processing applications for these mobile devices, what programmers should do is not only porting applications for general-purpose systems to these mobile platforms, but also implementing functions which determine processing precision and regulate throughput rate. Hence, programmers are required to write video processing applications for mobile devices and general-purpose systems separately and unportably, and it is troublesome for programmers to write such complicated programs.

On the other hand, CUDA[1] has been developed as an integrated development environment for GPUs. GPUs can achieve high performance by executing massively parallel threads simultaneously. Generally, video/image processing has data parallelism in its algorithm. For example, pixels in an image typically have data parallelism. So, image processing with GPU is likely to achieve high performance. However, for using GPUs effectively, a deep knowledge about them has been required. For example, data should be transferred between CPU and GPU, several memories of GPU which have different access speeds and sizes should be used according to the cases of processing, and a lot of threads should be managed. This would have been a burden to programmers.

To solve these problems, we have proposed a high-level video processing library RaVioli (Resolution-Adaptable Video and Image Operating Library) which guarantees pseudo real-time video processing on general-purpose computing systems. RaVioli conceals two types of resolutions, frame rate and the number of pixels, from programmers. RaVioli also simplifies the traditional iterative processing which used to be implemented by multiple loops. This makes image and video processing programming more intuitive. Furthermore, RaVioli has been improved to support CUDA GPU platforms. This improvement allows programmers to use GPUs effectively without considering GPU architectures.

However, because these functions are provided as C++ library, programmers have to consider the C++ language restrictions or features which are not essential for image processing. Hence, this paper proposes a new programming language as a front-end of RaVioli. With this language, programmers can describe image/video processing programs more intuitively. Furthermore, we have implemented a translator. It can generate both a program which can be linked with RaVioli and a program which can be linked with CUDA-supported RaVioli from a program written in this language.

With the translator, programs written in the language proposed in this paper are able to be executed on GPUs.

II. Research Backgrounds

A. Related Works

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 that has been proposed is the imprecise computation model (ICM)[2], [3]. 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 a suitable routine dynamically and empirically among them.

VIGRA[**?**] 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.

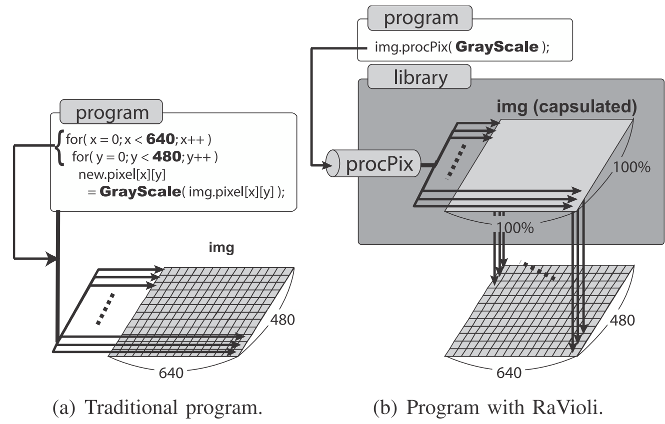
In addition, some libraries for image processing with graphics hardwares are also proposed. OpenVIDIA[6] is a library aimed at providing a GPU-accelerated processing framework for image processing and computer vision. It provides a simple API for some common computer vision algorithms. GpuCV[7] is an extension of OpenCV. It transparently manages hardware capabilities, data synchronization, activation of low level GLSL (OpenGL shading Language)[8] and CUDA programs, and offers a set of GPU-accelerated operators for image processing. MinGPU[9] contains all of the necessary functions to convert an existing CPU code to GPU code. It also implements several well known computer vision algorithms. Although these libraries help programmers to write image processing with GPU, it is impossible for programmers to write GPU programs without any knowledge about GPU or considering GPU architectures.

On the other hand, some programming languages for image processing are also developed. A loopless image processing language[10], for example, allows programmers to develop image processing programs for embedded devices without knowledge about the processors or memory architectures. This language enables to operate arrays without using loops in programs with some special operators. However, programmers have to write programs with a formula editor and consider array sizes.

B. Overview of RaVioli

The approach of the library RaVioli[11], [12] is completely different from existing computation models or image/video processing libraries/languages. RaVioli proposes a new programming paradigm with which programmers can write image/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 for motion object detection programs on computer systems. Such motion object detection programs are sometimes implemented by using a block matching algorithm, which searches for 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 another alternative method, and the method should be implemented by cumulative pixel value differences. Resolutions



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

Fig. 1. Digital image processing.

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 the innermost iteration, and the process is repeated for every pixel by loop nests as shown in Fig. 1(a). On the other hand, with 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 Fig. 1(b). GrayScale() in Fig. 1(b) is the routine defined by the programmer. What programmer should do are defining a function which processes one pixel and passing the function to one of the image instance's public methods. In this example, the method is procPix(). It is defined as a higher-order method, and applies a function, which is 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 as shown below.

1) One-to-one mapping.
   Processes one pixel and stores the result to the pixel. Used for grayscale, binarization, and so on.
2) Many-to-one mapping.
   Processes one pixel and its surrounding pixels, and stores the result to the centric pixel. Used for emboss filter, edge detection, and so on.
3) Many-to-many mapping.
   Processes objective pixels or partial images and get some value as the result. Used for template matching, and so on.

These higher-order methods apply the process for a pixel (or a partial image) to the entire image. But while writing these image processing programs, programmers have to select appropriate higher-order methods for required iteration pat-

```
1 Func Gray(img_a){
2   p1@img_a{
3     Ave = (p1.R + p1.G + p1.B) / 3;
4     p1.{R, G, B} = {Ave, Ave, Ave};
5   }
6 }(img_a)
```

Fig. 2. A grayscale program.

terns, and use them. This will be a burden for the programmer. Hence, this paper proposes a new programming language which provides a more intuitive programming paradigm for programmers.

III. Concept of the Programming Language

In this section, the common format of the programming language is explained.

A. Expression for Kernel Iterations

Generally, an image processing program has some kernel loops each of which is for iterating a process for a component part, such as a pixel or a small window. Therefore, with this language, programmers only have to describe some routines for the component parts and ranges where the routines are applied repeatedly. Although this concept is similar to RaVioli, programmers have to select a suitable higher-order method for each routine when using RaVioli. Hence, this paper proposes the following expression for describing a kernel loop with specifying the processing entity and processing object, or range.

Kernel Loop

```
processing entity @ processing object {
    kernel body
}
```

This allows programmers to describe any processing patterns like "pixel @ image" and specify the unit of the process and the region where the process is applied repeatedly in common format. Definition of data type is unnecessary and the variables in the block are basically handled as local variables.

B. Language Specifications

The specifications of the language will be explained by giving some specific examples of processing patterns discussed in section II-B, and showing programs which are written in the language.

1) One-to-One Mapping: The basic components of programs are definitions of the function name, the arguments, the calculations and the return values. When defining a function with this language, there should be "Func" before function name instead of the type of the return value. Fig. 2 shows a grayscale program with the language. There are particular variables such as IMAGE variable and PIXEL variable in the language. These variables are associated with an RV_Image

```
1  Func Edge_detect(img_a){
2    threshold = 127;
3    p1@img_a{
4      tmp = <−1, −1>p1.R + <0, −1>p1.R + <1,− 1>p1.R
5            + <−1, 0>p1.R − 8 ∗ p1.R + <1, 0>p1.R
6            + <−1, 1>p1.R + <0, 1>p1.R + <1, 1>p1.R;
7      if(tmp > threshold) p1 = #black;
8      else p1 = #white;
9    }
10 }(img_a)
```

Fig. 3. An edge detection program.

instance and an RV_Pixel instance, which are explained in section II-B. They are written in the formats as below.

PIXEL variable

p*n* (*n* is an integer number)

IMAGE variable

img_*a* (*a* is a string of alphanumerics.)

At the 3rd line, the average of each value of RGB is calculated, and the average is assigned to "p1" at the 4th line. The variable "p1" represents a PIXEL variable. The red color element of a PIXEL is represented by "R" concatenated by "." with PIXEL, such as "p1.R" at the 3rd line. Using a brace enables to handle values as a tuple. An expression beginning with "img_" is an IMAGE variable to handle all the pixels of the image and its information such as width and height. The expression "p1@img_in" at the 2nd line means that the processing object is referred as img_in and the processing entity p1 is the variable which represents an element of img_in. The process for p1, which is described in the function body, is applied to each PIXEL element in the img_in.

In this way, programmers can designate the processing entity and object easily using these expressions. Furthermore, the programmers need not to consider the iteration counts, or the number of pixels in the image.

2) Many-to-One Mapping: Fig. 3 shows an edge detection program written in this language. From the 4th line through the 6th line, a value which is calculated by using p1 and its circumferences is assigned to the variable tmp; the calculation is subtracting eight times R value of p1 from the sum of R values around p1. At the 7th and 8th line, if "tmp" is larger than "threshold", the color of p1 is set as black, otherwise as white. The expression "⟨x, y⟩" represents a set of coordinates, where x and y are the integer numbers. Using this expression allows programmers to handle relative coordinates. For example, "⟨-1,-1⟩p1" is an upper left pixel of the processing target element p1. The expressions "#black" and "#white" will be explained in next section.

3) Many-to-Many Mapping: Fig. 4 shows a template matching program written in this language. From the 1st line through the 3rd line, the variable declared in "GLOBAL" block is handled as a global variable and the other

```
1  GLOBAL{
2    img_tp;
3  }
4
5  Func TPmatching(img_a){
6    min = INT_MAX;
7    [img_tp]img_window@img_a{
8      (p1, p2)@(img_window, img_tp){
9        sum += abs(p1 − p2);
10     }
11     if(sum < min){
12       min = sum;
13       <x, y> = img_window.{x, y};
14     }
15     sum = 0;
16   }
17   img_a > writeBox(#red, <x, y>, [img_tp]) > img_b;
18 }(img_b)
```

Fig. 4. A template matching program.

variables are handled as local variables. The expression "[img_tp]img_window@img_in" at the 7th line means that processing entity is img_window whose size is as same as the IMAGE which is referred by a global variable img_tp, and processing object is img_in. The difference between a partial image img_window and the global template image img_tp is calculated from the 8th line through the 10th line, and a point where the difference is the smallest is searched for from the 11th line through the 14th line.

A special function "writeBox" is applied to img_a at the 17th line. Here, img_a is input to writeBox() by the symbol "⟩", and the result is output to img_b. This function "writeBox" is a predefined function which draws a rectangle on an image, and "writeLine" is also a predefined function which draws a straight line on an image. The synopses of these functions are shown below.

Predefined functions.

writeBox(#colorname, ⟨x,y⟩, [img_a])
writeLine(#colorname, rho, theta)

With writeBox function, it is possible to draw a rectangle which has size of the img_a and the coordinate of the rectangle's upper left corner is ⟨x,y⟩. The writeLine function is used in case of drawing a straight line by calculating the straight line expression using rho and theta. Here, "#colorname" means the color of the rectangle or the straight line. It is common to use the RGB model to express the color on computers, but the RGB model is not intuitively for users. For example, even if (R, G, B) is set as (D2, 69, 1E), users hardly understand that this means brown immediately. Therefore, the new language provides the COLOR variable for expressing the colors intuitively.

COLOR variable.

#white or #FFF

It is possible to describe the colors by name based on CSS3 color module[13], or hexadecimal number of triple digits after "#". For example, "#white", "#ABC", "#123" is handled as RGB values of (ff, ff, ff), (aa, bb, cc), (11, 22, 33) in hexadecimal number respectively.

4) Miscellaneous Expressions: Fig. 5 shows the main function of the template matching program shown in Fig. 4. The variables beginning with "file_" are the arguments for input/output files. At the 2nd and 3rd line, the input files are read into IMAGE variables img_in and img_tp. At the 4th line, a function "TPmatching" is applied to the processing object img_in, and an output image img_out is returned as the return value of TPmatching function. At the 5th line, the img_out is written into the output file.

```
1  Func main(file_in, file_tp, file_out){
2    img_in <= file_in;
3    img_out <= file_in;
4    img_in > TPmatching > img_out;
5    img_out => file_out;
6  }
```

Fig. 5. Main function for the template matching program shown in Fig. 4.

```
1  int _R_Gray_Ave;                                    // ... (a)
2  void Gray_loop0 (RV_Pixel* p1){                     // ... (b1)
3    _R_Gray_Ave = ((p1−>getR()) + (p1−>getG()) + (p1−>getB())) / 3;
4    p1 −> setRGB(_R_Gray_Ave, _R_Gray_Ave, _R_Gray_Ave);
5  }
6  void Gray (RV_Image* img_a){                        // ... (b2)
7    _R_Gray_Ave = 0;
8    img_a −> procPix(Gray_loop0);
9  }
10
11 RV_Image* img_in;
12 int main(int argc, char* argv []){
13   file .readBMP(argv[1],img_in);
14   Gray(img_in);
15   file .writeBMP(argv[2],img_in);
16   return 0;
17 }
```

Fig. 6. A converted grayscale program.

IV. Implementation of the Translator

In this section, we propose a translator, which translates a program written in the new language into both a program which can be linked with RaVioli and a program which can be linked with CUDA-supported RaVioli.

A. Translation into RaVioli Programs

Fig. 6 shows a RaVioli grayscale program which is converted from a program written in the new language. Here, all variables are declared as global variables with "_R_(function name)_" (a). The routine which is applied repeatedly to entire images (b1) and the routine which is applied at once such as initialization of variables (b2) are defined as different functions. By receiving the function gray_loop0, which is defined at (b1), as the parameter of the higher-order method

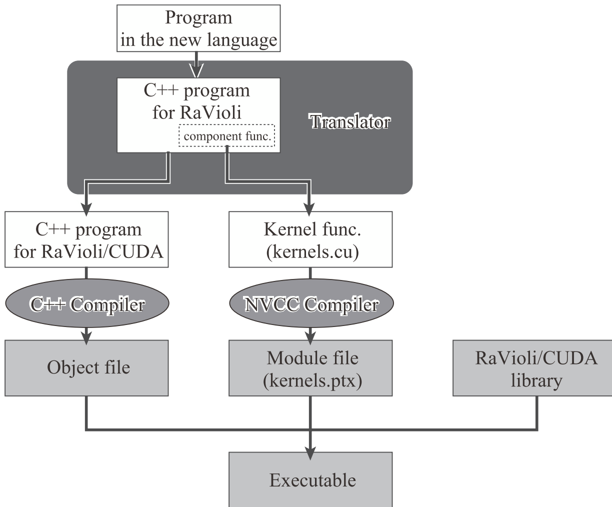


Fig. 7. Compilation flow with translator.

procPix() in function (b2), the calculation by the function (b1) is applied to the entire image "img_in".

B. Translation into RaVioli/CUDA Programs

CUDA-supported RaVioli (RaVioli/CUDA)[11] can provide an easy-to-use CUDA programming framework for developers. The translator can also generate RaVioli/CUDA programs from the programs which are written in the new language proposed in this paper. The generator of RaVioli/CUDA program is implemented as a postprocessor of the translator mentioned in IV-A. Fig. 7 shows a compilation flow with the translator.

The translator converts a program in Fig. 6 further to two program files; `main.cpp` for the host CPU and `kernels.cu` for the GPU device. These program files are compiled by C++ compiler and CUDA compiler `nvcc`, and assembled to an executable. The results of this translation are shown in Fig. 8 and Fig. 9.

In the main program, the invocation of procPix() in Fig. 6 is converted to cudaProcPix() in Fig. 8. A statement of GetKernelHandler() is added in main() for getting a kernel handler for the component function Gray(). 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 Gray() is converted to a kernel function. A kernel function expresses a process for one thread. In Fig. 9, the kernel function Gray() is defined as it processes one pixel on one thread. The definition of Gray() also makes continuous threads process continuous pixels by calculating indices. This is for coalesced access of CUDA memories. Memory accesses to the global memory by continuous sixteen threads in each *Block* can be issued in parallel owing to this code conversion.

The translator searches for 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 needs no

```
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, "Gray");
9    image.cudaProcPix(&cuFunction);
10    :
11   device.RaCudaExit(); /*  finalize device */
12 }
```

Fig. 8. Main program translated from Fig. 6.

```
1  /* kernels.cu */
2  extern "C" __global__ void
3  Gray(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 }
```

Fig. 9. Kernel program translated from Fig. 6.

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. Detailed discussion about reduction operation in RaVioli is described in [11].

Enumerating translation rules in detail is left out for want of space, but 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 generated 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 the heap memory, and the other is the page-locked host memory. The 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.

TABLE I
EVALUATION OF PROGRAM SIZE.

Programs	*Program Size*	*Native C++*	*RaVioli*	*New Language*
Grayscale	lines	38	22	12
	bytes	910	486	244
Emboss filter	lines	153	26	15
	bytes	4074	867	444
Template matching	lines	114	50	21
	bytes	3946	1175	430
Line detection	lines	139	79	50
	bytes	3518	2163	1084

V. EVALUATIONS

The new programming language and the translator described in III and IV were evaluated with several image processing programs.

A. Evaluation of Program Size

On programs with native C++, traditional RaVioli and the new language, the program size are compared by linage and bytes. TABLE I shows that the linage and bytes of typical four programs written in the new language are smaller than that of programs with native C++ and traditional RaVioli.

B. Evaluation of Readability

Binarization programs written in native C++, traditional RaVioli and the new language are shown in Fig. 10, Fig. 11 and Fig. 12. The readability of a program in the new language is compared with that of the programs with C++ and RaVioli. Both RaVioli and the new language conceal the resolutions and the number of components from programmers. So, programmers can be released from resolutions and the number of iterations. For applying the user-defined function to all processing entities, programmers are required to select an appropriate higher-order method with RaVioli; in this program, the higher-order method is "procPix." But with the new language, programmers can be unaware of it. As just described, programmers only need to write essential processes, and readability is increased.

C. Performance Evaluations

We have also evaluated the computation performance. The evaluation environment is shown in TABLE II, and the evaluation results are shown in TABLE III. In TABLE III, *C++* denotes programs with native C++, *RaVioli* denotes programs with traditional RaVioli, *Generated RaVioli* denotes programs with RaVioli translated from programs in the new language and *Generated RaVioli/CUDA* denotes programs with CUDA-supported RaVioli translated from the new language program. The size of the image which was used for grayscale and emboss filter programs was 512×512 pixels and for line detection program was 450×540 pixels. For template matching program, the base image has 395×372 pixels and the template image has 70×72 pixels.

```
1  void Binary(Img* img){
2    int i, j, v, tmp;
3    for(i = 0; i < img -> height; i++){
4      for(j = 0; j < img -> width; j++){
5        v = getV(img -> D[i][j]);
6        if(v < 85) tmp = 255;
7        else tmp = 0;
8        img -> D[i][j].r = tmp;
9        img -> D[i][j].g = tmp;
10       img -> D[i][j].b = tmp;
11     }
12   }
13 }
14 Binary(&Image);
```

Fig. 10. A binarization program in native C++.

```
1 void Binary_Pix(RV_Pixel* p){
2   if((p -> getV()) < 85) p -> setRGB(225, 225, 225);
3   else p -> setRGB(0, 0, 0);
4 }
5 Image -> procPix(Binary_Pix);
```

Fig. 11. A binarization program with traditional RaVioli.

```
1 Func Binary(img_a){
2   p1@img_a{
3     if(p1.V < 85) p1 = #black;
4     else p1 = #white;
5   }
6 }(img_a)
7 img_in > Binary > img_out;
```

Fig. 12. A binarization program in the new language.

As we can see in TABLE III, there is little difference between *RaVioli* and *Generated RaVioli*. This means that the translator which converts a program in the new language into RaVioli doesn't output useless statements, and converts appropriately. *Generated RaVioli/CUDA* achieves performance gains of 3.9-fold, 7.8-fold and 135.4-fold on grayscale, emboss filter and template matching respectively, over traditional RaVioli programs without rewriting programs. Typically on template matching, *Generated RaVioli/CUDA* also achieves about 38.7-fold speedup over *C++*.

The performance of *Generated RaVioli/CUDA* on the line detection program is not evaluated, because the translator does not support this type of program at present. The hough transform in line detection is a little different from other general image processing. Although both the input and output of general image processing are images respectively, the input of the hough transform is an image and the output is an array. Hence, there is a case where an element of an array can be written multiple values calculated from different coordinates.

TABLE II
EVALUATION ENVIRONMENT.

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 III
EXECUTION TIME (MS).

Programs	*Native C++*	*RaVioli*	*Generated RaVioli*	*Generated RaVioli/ CUDA*
Grayscale	1.15	4.81	5.06	1.21
Emboss filter	4.77	10.15	9.87	1.30
Templ.matching	2425.65	8483.45	8173.30	62.62
Line detection	35.32	71.02	89.65	-

Therefore, a simple parallelization for this type of processing causes data conflicts and incorrect results. If there is a data dependency, some reduction operations have been necessary to solve this problem. But in this case, the amount of memory is insufficient for giving all arrays to each thread. This is why the line detection program is unable to be converted to CUDA-supported RaVioli program at present.

VI. CONCLUSIONS

This paper proposes a programming language which allows programmers to write image processing intuitively. Generally, an image processing program has some kernel loops each of which is for iterating a process for a component part. Programmers only need to describe some routines for the component parts with this language. It allows programmers to be unaware of pixels and resolutions. The routine is applied to the entire image automatically through the higher-order methods of RaVioli. RaVioli is a pseudo real-time video processing library which conceals spatial/temporal resolutions from programmers and changes resolutions automatically. With the language proposed in this paper, image processing programs can be described more intuitively than RaVioli programs. In this paper, a translator is also proposed. The translator converts the programs written in the new language to RaVioli programs, and converts the translated RaVioli programs further to CUDA-supported RaVioli programs. With this translator, several programs written in the new language such as grayscale, template matching and line detection programs are able to be converted to the programs with RaVioli and CUDA-supported RaVioli. Automatic translation into CUDA-supported RaVioli programs enables programmers to use GPUs effectively without considering GPU architectures.

Possible improvement of this study is improving this language to be able to handle more types of image/video processing. Now, RaVioli has good writeability, and many programs such as edge detection, circle detection, hough transform, and so on can be written with RaVioli. But the new language cannot describe all of these programs at the moment. In addition, this language doesn't correspond to description of video image processing.

APPENDIX
AN EXAMPLE OF TEMPLATE MATCHING PROGRAM

Another example of code conversion by the translator is shown in this appendix. The codes shown in Fig. 13 and Fig. 14 are template matching programs which are generated by the translator from the code in Fig. 4.

From the 8th line through the 10th line in Fig. 4 is converted to the TPmatching_Comp() in Fig. 14, and the function TPmatching() is converted to the TPmatching_kernel() in Fig. 14.

In the main program shown in Fig. 13, TPmatching() is defined. It gets kernel handlers for kernel functions, sets up texture reference, and passes kernel functions to cudaProcBox(). The cudaProcBox() is one of the higher-order methods of RV_Image instance, and it is for applying a component function repeatedly inside a certain box.

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

Thread-local results are stored in the data4reduction[] array. In Fig. 4, the variable *sum* is overwritten in the function TPmatching(). This lets the translator know that there needs a reduction operation for the variable *sum*. Hence, the code for reduction shown in Fig. 15 is also generated.

The code in Fig. 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.

ACKNOWLEDGMENT

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] NVIDIA Corp., *NVIDIA CUDA Programming Guide*, 2nd ed., Jun. 2008.

[2] 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.

[3] 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. of the 17th Int'l. Conf. on Pattern Recognition (ICPR'04)*, vol. 1, 2004, pp. 736–740.

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

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

```
#include " ravioli .h"
#include " cutil .h"

RV_Cuda device;

CUtexref cuTexTPref; // texture reference for template image
CUarray d_TPimage;
int3 result;

void TPmatching(RV_Image* image){
  CUfunction cuFunction;
  CUfunction cuFunction2;
  device.GetKernelHundle(&cuFunction, "TPmatching_kernel");
  device.GetKernelHundle(&cuFunction2, "reduction_kernel");
  cuParamSetTexRef(cuFunction,
                   CU_PARAM_TR_DEFAULT,
                   cuTexTPref);
   result = image->cudaProcBox(&cuFunction,
                               tp_image->Width,
                               tp_image->Height,
                               &cuFunction2);
}

int main(int argc, char* argv []){
  RV_Image* input_image = new RV_Image(argv[1]);
  RV_Image* tp_image = new RV_Image(argv[2]);

  device.RaCudaInit();
  device.GetTexrefHundle(&cuTexTPref,"texTP");

  tp_image->TexRefSetImage(&d_TPimage, &cuTexTPref);
  TPmatching(input_image);
  cutilDrvSafeCall(cuArrayDestroy(d_TPimage));
  image->writerect(result.x, result.y);

  device.RaCudaExit();
  return 0;
}
```

Fig. 13. Main program translated from RaVioli program.

[6] J. Fung, S. Mann, and C. Aimone, “OpenVIDIA: Parallel GPU Computer Vision,” in *Proc. the ACM Multimedia 2005*. ACM, 2005, pp. 849–852.

[7] Y. Allusse, P. Horain, A. Agarwal, and C. Saipriyadarshan, “GpuCV: An Opensource GPU-Accelerated Framework for Image Processing and Computer Vision,” in *Proc. 16th ACM international conference on Multimedia*. ACM, 2008, pp. 1089–1092.

[8] R. J. Rost, *OpenGL Shading Language*. Addison-Wesley professional, 2004.

[9] P. Babenko and M. Shah, “MinGPU: a Minimin GPU Library for Computer Vision,” *IEEE Journal of Real-Time Image Processing*, vol. 3, no. 4, pp. 255–268, 2008.

[10] J. Segawa and T. Kanai, “The Array Processing Language and the Parallel Execution Method for Multicore Platforms,” *The First International Symposium on Information and Computer Elements*, 2007.

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

[12] 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.

[13] T. Çelik, C. Lilley, and L. D. Baron, “CSS Color Module Level 3,” W3C, Tech. Rep., Jun. 2011.


```
texture<int, 2, cudaReadModeElementType> texTP;
__device__ int TPmatching_Comp(int* idata, int wid, int hei,
                               int widBox, int heiBox, int x, int y){
  int sum = 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);
      sum += absDiff(p1, p2);
    }
  }
  return sum;
}

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 min = INT_MAX;
  for(int j = y; j < (hei - heiTP); j += incY){
    for(int i = x; i < (wid - widBox); i += incX){
      int sum = TPmatchi_Comp(idata, wid, hei, widBox, heiTP, i, j);
      if(sum < min){
        data4reduction[y * 256 + x].z = sum;
        data4reduction[y * 256 + x].x = i;
        data4reduction[y * 256 + x].y = j;
      }
    }
  }
}
```

Fig. 14. Kernel module program translated from RaVioli program.

```
extern "C"
__global__ void
reduction_kernel(int4* data4reduction, int4* g_odata){
  __shared__ int sdatax [256], sdatay [256], sdataz [256];

  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();

  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];
  }
}
```

Fig. 15. Reduction operations generated from Fig. 4.