IMGVM: An Image Oriented Virtual Machine for Real-Time Computer Vision

David Martínez Oliveira, Fernando Martín Rodriguez Xulio Fernández Hermida
ETSIT Communication and Signal Processing Deparment
University of Vigo
Vigo, Spain
Email: dmartin@uvigo.es, fmartin@tsc.uvigo.es, xuliofh@uvigo.es

***Abstract*— In this paper a simple architecture for an image oriented virtual machine targeted to computer vision applications is introduced. This virtual machine provides a low level approach to image processing working on very high level data structures which allows developers to setup the sequence of processing actions in an easy way. Additionally, this architecture provides a common environment for prototyping and production systems simplifying the deployment process of the final application.**

I. Introduction

Computer vision applications[5] involve several processing stages in order to solve a given problem. Independently of the application domain, any computer vision (CV from now on) applications implies, as a minimum, an image processing [3][4] stage where the source image captured with a given device is processed to make easier the work of later stages (parameter extraction, classification, etc...).

Image processing stage must be fast because there are much more things to do in a very short time (40 ms for typical video processing). Moreover, most of these systems must work in industrial environments where small and embedded appliances are required in order to integrate the CV system into the product line where it will work.

There are several systems and tools to build up this kind of systems, mainly nowadays, where embedded PC-based solutions can be used instead of full custom developments. In this last line, researchers can find tools like Matlab [1] which provides a very complete image toolbox or IDL [2] also used for image processing applications. But in real world, computer vision applications usually work on specific libraries (commonly associated to the chosen hardware for the application) and own code development using some middle level languages like C/C++. There are as many libraries as hardware vendors and some general solutions as the Intel Computer Vision Library (OpenCV)[9], which sometimes are cumbersome for the application being developed.

In this scenario, there are two main ways to carry out application prototyping. In the first one, researchers can work directly on the final system which, in general, makes implementation and testing a complex task involving editing, compiling and testing cycles in a low-level environment.

In the second one, researcher works with tools like the ones named above (Matlab, OpenCV) that, once the system is finished, need to be translated to the final application environment, which implies a new (but shorter) testing process. Moreover, running a tool like Matlab in and industrial embedded system will require a big amount of resources, normally not available in these kind of systems.

In general, tools that are good for prototyping are not good for final systems because they are in general slower that direct coding in an intermediate programming language.

From these two main choice, the second one has the additional advantage of providing researchers with an *interactive* environment where testing their application in an easy and fast way. This is important for solving new problems where some probe and error cycles are required to reach the good solution for that problem.

Therefore, a common solution to allow easy prototyping of CV systems that could be used directly on the final system, would be of great interest reducing the required work to develop and setup a computer vision application.

In this paper we propose a simple approach to image processing for real-time computer vision applications built on the idea of an image-based virtual machine. Researchers will use a very low level language similar to assembler but working in very high level data structures like images or regions of interest.

This system allows easy prototyping of systems writing small *assembler* programs based on a given set of op-codes which represent common image processing algorithms, in a similar way to scripts in software applications. These algorithms are implemented in a middle or low level programming language so they will run at high speeds. At the same time, the system is highly configurable so virtual machine parameters can be easily modified for prototyping operation and then reduced for the final system just editing a configuration file.

This paper is organised as follows. Section II describes the architecture of our image-based virtual machine. Section III shows how to manage this virtual machine. Section IV describes the virtual machine assembler language and finally, Section VI presents the current implementation and results achieved in this work.

II. IMGVM Architecture

The IMGVM (IMaGe Virtual Machine) was modeled against a typical register processor where each register can

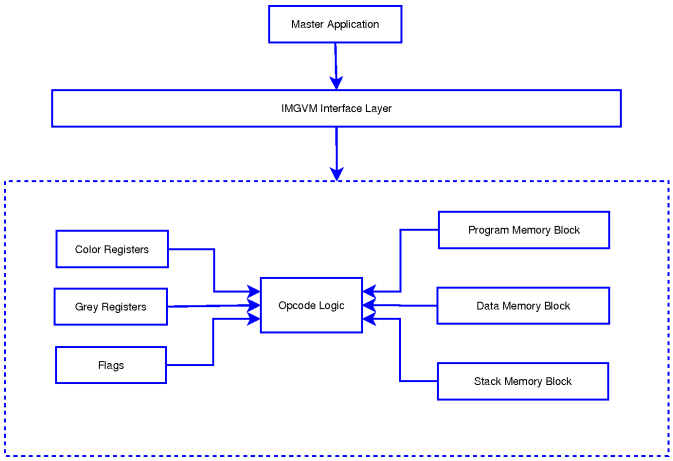

Fig. 1. IMGVM Block Diagram

contain complex data structures like images or regions of interest. Figure 1 shows the general IMGVM block diagram.

The different blocks shown in figure 1 are summarised below:

- An Instruction Pointer Register
- A set of gray-scale and color image Register
- A program memory block
- A flag register
- A data memory block
- A stack memory block
- A set of op-codes.

Any of the items of the IMGVM can be configured along with the program running in a given CV application. This includes the number of registers the machine will use, the size of the program, data and stack memory blocks and even the set of op-codes the virtual machine will understand.

IMGVM provides a set of default op-codes which provides very simplistic operations on registers (images), however, programmers can provide its own *opcodes* implemented as independent shared libraries, so the final application will only contain the op-codes it needs. Most of the basic image processing algorithms has been implemented this way and included in the system.

The virtual machine does not know anything about input/output tasks. This makes it very simple, and more important, multiplatform. Even when our current implementation uses C language and runs in GNU/Linux environments, the implementations doesn't make use of any system specific functionality making it highly portable.

After this general description of the virtual machine we will go deep in each component throughout next sections.

A. Registers

As introduced early in this text, the IMGVM registers are very high level object like images or regions of interest (ROI hereinafter). The data structures associated to each IMGVM register are fully open so any developer can add additional objects/datatypes like segments, points, vectors or whatever required for a given application. Adding further objects implies adding further op-codes able to work on them (see section IV-A below).

Current implementation only works on images and ROIs. In general the maximum size of images must be fixed after starting the virtual machine. It would be easy to modify the system to work with arbitrary size images but this approach makes the system more uniform, faster, and ROIs can be used to work with subimage data.

The system automatically manages color and grey scale images providing two separated sets of registers to work with each kind of data. So, developers can choose to work on color images or gray-level images to build up hybrid algorithms working on both kinds.

It will be easy to extend the system to support other kinds of images (1-bit monochrome, integer/float images, etc...), but, until now, none of these options was required.

B. Program Memory

It is in the program memory where the sequence of op-codes composing the program resides. The virtual machine has an special register called instruction pointer (IP) which, as usual, points to the current program memory position being executed, as usual.

Each program memory position has a fixed size able to contain different types of data. These types of data are listed below:

- Op-code: A valid IMGVM op-code from currently selected one.
- Register: A reference to a IMGVM register
- Integer: An integer value
- Float point number: A floating point value.

Several op-codes require some parameters which are stored sequentially in the program memory, and just after it. Op-code implementation makes the work of decoding these parameters and update the IP accordingly. Suitable macros are provided to implement parameter decoding in an easy way.

C. Flags

In the same way of every component of the IMGVM, system flags are also configurable but appropriated op-code should be provided in order to make them useful.

Default IMGVM configuration provides the following flags:

- ROI: used to indicate that a given operation generates ROI in a given register or modifies it.
- TYPE: used to indicate that a given operation was applied to a grey-level image or on a color image. In the case that the operation involves both types, the *bigger* one will be selected.
- OVERFLOW: used to indicate that a given operation produced an overflow in the registers it affected.

IMGVM provides generic conditional jump instructions to allow applications make use of these flags. Users adding their own flags should add appropriated instructions to use them.

III. IMGVM Interface Layer

This section describes the IMGVM programming interface details required to use it in CV applications.

A. Internal Data Structures

Due to the very simple design of the IMGVM, no input/output functionality is provided by the system then, in order to perform this kind of operation, IMGVM users must access internal data structures of the system. These internal data structures are fully encapsulated independent entities. Actually they can be used out of the IMGVM environment.

The main internal data structure is `GRTK_ITEM` which is in fact the type of the IMGVM registers. This structure can hold different data types which can be managed uniformly through a simple interface. Basic types of `GRTK_ITEM` structure are the following:

- `GRTK_IMAGE24`: A true-color image
- `GRTK_IMAGE8`: A grey-scale image
- `ROI24`: a true-color region of interest
- `ROI8`: a grey-scale region of interest.

Additionally, each `GRTK_ITEM` has two important fields from the programmer point of view. First one is the `subitems` field which holds other `GRTK_ITEM` items which are parts of the main item. For instance, regions of interest within a given image are stored in this field.

The other important field in `GRTK_ITEM` is the `metadata` field. This field is a generic vector which can be used to store extra information in a given item which can be used later in a IMGVM program. This field can also be used to transfer data from the virtual machine to the main application in a very simple way.

B. IMGVM Interface

Virtual machine control and configuration interface is located in the IMGVM Interface layer. This interface allows programmer to configure each IMGVM parameter, to load op-codes lists, to transfer data to/from virtual machine/host application and to load and run IMGVM programs. Section V-B shows the use of this interface. Detailed description of it is out of the scope of this paper and is covered in [8]

IV. IMGVM Assembler

The IMGVM provides a simple built-in assembler to inject code within the virtual machine from an external text file. Op-code mnemonic code is composed of a reserved keyword plus a variable set of parameters. Parameters can be integer numbers, floating-point numbers or virtual machine registers.

Virtual machine registers are named as follows. Color registers begins with character 'c' followed by a number that indicates the register number. Grey-level registers begins with character 'g' followed by a number indicating the register number.

Assembler source code can include labels as usual. Simple strings followed by the ':' character.

There are two special reserved keywords:

```
int grtk_op_sadd (GRTK_IMGVM vm, GRTK_IMGVM_OPCODE *p)
{
  GRTK_ITEM           o1, o2;
  int                 off, size, val, s;
  unsigned char       *a, *b;

  GRTK_OPCODE_START(p);  /* START Opcode Macro*/

  /* Opcode Decoding SOURCE_REG, TARGET_REF, SCALAR_VALUE */
  o1 = GRTK_DECODE_IMG(vm);
  o2 = GRTK_DECODE_IMG(vm);
  s  = GRTK_DECODE_INT(vm);

  /* Get pointers to image data and image buffer size */
  a = (unsigned char *) o1->data;
  b = (unsigned char *) o2->data;
  size = o1->w * o1->h * (o1->type == GRTK_ITEM_IMAGE24 ? 3 : 1);

  /* BEGIN: Real opcode implementation */
  for (off = 0; off < size; off++)
    {
      val = *(a + off) + s;
      *(b + off) = (val > 255 ? 255 : val);
    }
  /* END: Real opcode implementation */
  GRTK_OPCODE_RETURN; /* END Op-code Macro*/
}
```

Fig. 2. Scalar Addition Op-Code Implementation. Example of instruction decoding using helper macros

- `END` that indicates the end of the code and will stop the virtual machine execution
- `DATA` that indicates that that program memory position contains data, typically op-code parameters.

A. Op-codes

IMGVM provides a default set of basic built-in op-codes which perform common operations on images. However, IMGVM users can provide their own set of op-codes for a given application as a shared library that can be included within the virtual machine directly.

As mentioned before, implementation of IMGVM op-codes implies the implementation of the op-code decoding which can be done easily using the helper functions and macros provided by the system. Figure 2 shows a simple example on how to implement a simple `SADD` (scalar addition) instruction which adds an scalar value to every pixel in a given image and stores result in a given target image.

As can be seen in Figure 2 implementation of new op-codes is very simple using the helper macros provided by the system. Op-code implementation functions receive two pointers. First one points to the virtual machine itself so the code can access registers and flags in order to modify them. The second pointer points to the program memory address of the instruction being decoded. This parameter allows to execute a given op-code out of the virtual machine program memory block and can be used for debugging purposes.

Table I summarises currently implemented op-codes in IMGVM.

V. Use Case

In this section a real system using the IMGVM is described. It is a simple optical tracking system based on color segmentation and was developed as part of a bigger project on gesture recognition for virtual reality environments.

```
; GRTK Optical Tracking System. Image Processing Stage
; (c) GPI-RV, 2006
;

; Color segmentation of captured image and thresholding
COLOR_ANN_CLASSIFIER c0 c1 0
COLOR2GREY c1 g1
GREY_THRESHOLD g1 g4 100

; Locate Regions of Interest to speed up processing
ROI g4 g5
JNROI FINISH   : Jump if NO ROI

; Remove background noise and try to remove holes in object
; Opening
ERODE g4 g5 2
DILATE g5 g6 2
; Extract Parameters.
; Classifier expects stat parameters in g2
COPY g5 g3
STATS g3 g4
; Display stuff for debugging
CORRECT_ROT g3 g4
DILATE g4 g3 3
ROI g3 g2
DRAW_ROI g3
FINISH:
END
```

Fig. 3. Optical Tracking System IMGVM code. This example locates the user hand, computes image moments and stores them for further processing by main application

A. Optical Tracking

In this application, real-time video must be processed in order to extract the position of the user's hand or some kind of pointing device. For real-time virtual reality interaction a quite high processing frequency is required in order to provide a realistic sensation to users.

The IMGVM system was used to carry out the image processing stage of the system which consists of making the segmentation of the pointing device and the extraction of some basic data like position and orientation.

Figure 3 shows the basic code of the image processing stage for this optical tracking system.

The code is very self-explanatory since mnemonics where chosen to match common image processing algorithms. As can be seen in figure 3 the system performs a very simple color segmentation and immediately converts the segmented image to grey-scale in order to perform a faster processing.

Color segmentation (`COLOR_ANN_CLASSIFIER`) is carried out by a simple one-neuron perceptron neural network (actually it is managed as the simplest MLP[6]) which has been externally trained and whose parameters are injected in the virtual machine by the main application. Main application loads MLP parameters in data position 0 (third parameter) so different network configurations could be tested.

Then, the program looks for regions of interest in the image and finishes processing if no ROI was found. This is done because the rest of the processing sequence is very computationally expensive if the whole image must be processed (and not a smaller ROI).

Extracted information data is stored in the metadata field of register `g2` (this is done by the `STAT` op-code) where the main application expects to find them for transmission to the VR system.

TABLE I
CURRENT OP-CODES GROUPS INCLUDED IN IMGVM

Group	Description
Morphology	Basic Morphology operands (erode, dilate, ...)
Arithmetic	Arithmetic operation (sum, plus, diff, ...)
Logical	Logical operations (and, or, xor, ...)
Comparison	Comparison operations (gt, lt, eq, ...)
Statistics	Statistics (stat, k-mean, ...)
Color	Color Manipulation (grey, rgb, hsv, component ...)
Filters	Linear Filters, Edge Detection, Rank Filter, ...
Geometric	Geometric operations (rotation, scaling, ...)
Segmentation	Color/grey segmentation, ANN clasifiers ...

```
01  GRTK_ITEM    img, result;
    GRTK_IMGVM   vm;

    /* Initialize the IMGVM */
05  vm = grtk_imgvm_new (image_w, image_h);

    /* Get the GRTK_ITEM associated to color register 0*/
    item =  grtk_imgvm_get_reg (vm, COLOR, 0);

10  /* Load code in the virtual machine */
    grtk_imgvm_load_code (vm, "my_code.ivm");

    /* Main loop processing real-time images */
    while (1)
15  {
      image_data = get_image_from_video_src ();
      /* Update item with new data which is associated with c0 register*/
      grtk_item_img_set (item, image_w, image_h, image_data);

20    /* Run code in the virtual machine */
      grtk_imgvm_run (vm);

      process_data (vm);
    }
```

Fig. 4. Optical Tracking System Main Application. Using IMGVM high level interface from a real CV application

B. Main Applications

The main optical tracking application starts the virtual machine and loads code on it. Then begins acquiring images from a video source and, for each captured image, the program in figure 3 is executed and processed. Extracted information is sent through the network to a render station where a pointer object is drawn in the virtual environment.

Figure 4 shows the parts of the main applications related to IMGVM.

The first thing to point out is the simple interface provided to upper level applications. This simple interface makes very easy the use of the system at the same time that keeps image processing stage totally uncoupled of the whole system. The only exception is the `process_data` function.

The second question to point out is the way data are inserted within the virtual machine. As said above in the text, IMGVM does not provide any input/output facility, so it is responsibility of the higher level of the application to inject data in the virtual machine.

This process is carried out accessing the virtual machine registers directly (line 8 of figure 4). In the main loop the application injects data acquired from the video source in this internal data struct (`GRTK_ITEM`) before executing the code in the virtual machine (line 21).

The `process_data` function also needs to access these data structures in order to extract information from a given register metadata or from somewhere else in the data memory block.

VI. Implementation and Results

The system described along this paper has been implemented for the GNU/Linux operating system and used successfully in the application described in section V for optical interfacing to virtual reality environments. Figure 5 shows two screenshots of the application at work.

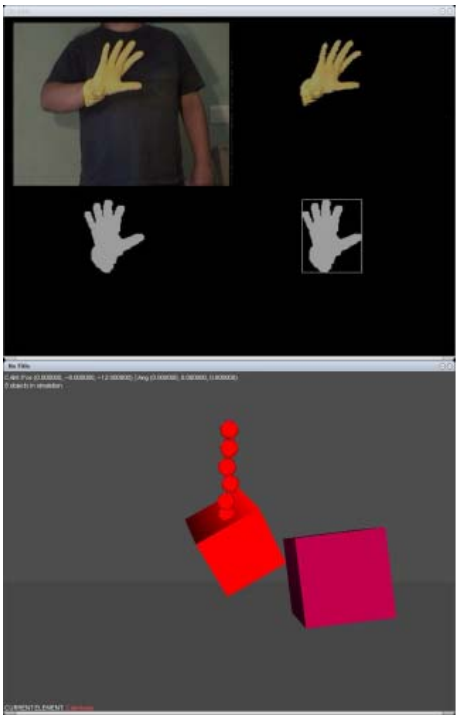

Fig. 5. Up image shows the IMGVM training application at work. Down Image shows the test virtual reality environment

Description of the virtual reality environment is beyond the scope of this paper. The unique remarkable comment about it is that it runs in a different machine and receives interaction commands from the IMGVM system from the network.

The optical tracking system was developed using the IMGVM. It has been integrated in a bigger application that provides a suitable virtual reality user interface. This IMGVM Test Application (ITA) does the following actions:

- Image capture and injection in the embedded IMGVM
- Visualization of images stored in internal IMGVM registers
- Extraction of data computed by the IMGVM programs
- Interpretation of these data to generate tracking and gesture recognition information
- Transmission of this information to the remote VR engine through the network

In this application the IMGVM virtual machine was statically linked to a general training application using OpenGL for rendering (this is only an user helper feature not required for a final implementation). The up part of figure 5 shows the main screen of the ITA (*IMGVM Test Application*).

Its final size on disk is 76 Kb and spends less than 15ms in processing 256x256 images in a AMD K7 1GHz processor. These values make it suitable for embedded applications. At the same time it keeps the simplicity of developing in desktop computers.

VII. Conclusions

In this paper we have presented a simple approach to real-time video processing based on a high-level abstraction of a typical processor architecture. Our system is completely configurable in order to adapt it to different deployment platforms, running from embedded systems to high-end computers.

Our current implementation, presented along this paper, provides a simple environment to prototyping CV applications based on well-known image processing algorithms. At the same time, provides an easy framework for implementing new implementations (op-codes) and quick testing of them, using the IMGVM Test Application (ITA).

This approach provides a single development environment with the required simplicity and interactivity for prototyping development. At the same time, it allows to directly translate that prototype to the final real system.

The key point in the system described is the duality between high and low level development.

On one hand the system is programmed in a very low-level assembler language that allows engineers to concentrate on the problem to resolve at the level of well-known image processing algorithms (op-codes). This is the classical approach for this kind of applications.

On the other hand the virtual machine abstraction is defined at a very high level where machine registers hold complex data structures (encapsulated to developer) and machine op-codes represent complete algorithms instead of basic operations at the machine level.

A real optical tracking system was developed and tested in a real VR environment, and was improved to its use in a distributed environment common in VR deployments.

Finally, the simplicity of the approach and its hardware-like design introduces a new research line about real-hardware implementation using modern programmable devices. From this point of view, the presented approach will allow high performance hardware implementations in a simple way, closing the gap between software and hardware for computer vision applications.

VIII. Future Research

The main research line should be the implementation of the IMGVM directly in hardware. The simple design of the

solution exposed in this text makes easy its implementation in custom hardware.

From this point of view, the separation of op-codes from the virtual machine itself provides a clear block separation for an FPGA implementation. So, hardware engineers could convert each op-code implementation in an FPGA core which will allow simple configuration. Furthermore the same machine code used in the software version can be feed in the hardware implementation obtaining the same results.

This possibility will make the development of computer vision applications much easier since main development would be carried out in a normal computer and then be easily translated to different embedded solutions ranging from small computers (PC-104[10]) to full hardware implementation (FPGAs) assuming there is an VHDL[7] implementation of the used op-codes.

So future research lines will focus in two main topics:

- Basic hardware implementation of the virtual machine
- Automatic procedures to translate op-code implementations to VHDL in order to easily plugged to a programmable device.

The main issue with a hardware implementation of this virtual machine is memory amount. Current programmable devices do not allow to define big enough RAM/Registers units to hold images. Maybe future devices would provide such capacity, but meanwhile, a hardware implementation of IMGVM should use external memory to hold its registers. Op-code, as pointed above, will be deployed like cores and FPGA reconfiguration facilities will provide the required infrastructure to simulate shared libraries on our software implementation.

Note that hardware implementation of image processing algorithms can be improved due to the inherent parallelism of them, allowing the hardware system to work simultaneously in several parts of the image.

Both lines are in an early stage at this moment.

Other interesting research line is improving the system interface to generate op-codes from matlab code allowing researchers to keep working on Matlab, but providing an easy way to translate their algorithms to real systems using our IMGVM.

REFERENCES

[1] Rafael C. Gonzalez, Richard E. Woods, Steven L. Eddins. *Digital Image Processing Using MATLAB,* Prentice Hall; 1st edition 2003

[2] Liam E. Gumley *Practical IDL Programming,* Morgan Kaufmann; 1st edition, 2001

[3] Rafael C. Gonzalez, Richard E. Woods. *Digital Image Processing* Prentice Hall; 3rd edition 2006

[4] Jain, Anil K. *Fundamentals of Digital Image Processing*. Englewood Cliffs, New Jersey: Prentice Hall, 1989

[5] Shapiro, Linda G., Stockman, George C. *Computer vision* Prentice Hall, 2001

[6] Haykin, Simon S. *Neural networks : a comprehensive foundation*. Englewood Cliffs (New Jersey) : MacMillan, cop. 1994

[7] Volnei A. Pedroni. *Circuit Design with VHDL*. The MIT Press (2004)

[8] D. Martínez, "IMGVM Specification 2.0," unpublished. Technical Report available at `http://wgpi.tsc.uvigo.es/tech_reports`

[9] Intel Corp. "Open Source Computer Vision Library" unpublished. `http://www.intel.com/technology/computing/opencv/`

[10] PC-104 Embedded Consortium. "PC-104 Specification Version 2.5" unpublished. `http://www.pc104.org/technology/pc104_tech.html`