OBJECT ORIENTED FRAMEWORK FOR REAL-TIME IMAGE PROCESSING ON GPU

Nicolas Seiller†, Nitin Singhal‡, and In Kyu Park†

†School of Information and Communication Engineering, Inha University, Incheon 402-751, Korea
‡DMC R&D Center, Samsung Electronics CO., LTD., Suwon 443-742, Korea

ABSTRACT

In this paper, we present a framework for efficiently integrating programming resources of both GPU and CPU. We introduce an object oriented framework for GPGPU-based image processing. We illustrate a set of classes exploiting the design and programming advantages of an object oriented language, such as code reusability / extensibility, flexibility, information hiding, and complexity hiding. This class structure is supplemented with shader (GLSL) and kernel (CUDA) programming to facilitate full functionality. We demonstrate the potential of our approach with application scenarios and discuss the framework's performance in terms of programming effort, execution overhead, and speedup factor achieved over CPU.

Index Terms— Object oriented framework, GPGPU, class hierarchy, GLSL, CUDA

1. INTRODUCTION

Efforts in general purpose computation on graphics processing unit (GPGPU) [1] have created a wealth of opportunities for developers to offload computationally intensive tasks to the graphics processing unit (GPU), which offers significant speedup compared to the central processing unit (CPU). Among diverse fields of GPGPU, image processing and computer vision have gained considerable attention. Image processing algorithms usually perform the same computation on a massive number of pixels or features. Therefore they can exploit the SIMD (single instruction multiple data) GPU architecture and therefore can be effectively implemented on the GPU in many cases.

Through development of elaborate interfaces, such as OpenGL Shading Language (GLSL) [2] and NVIDIA Compute Unified Device Architecture (CUDA) [3], the GPU can be used to process data and deal with computationally intensive tasks. However, these interfaces offer a rather complex integration framework and require much more than individual kernel programming. As a consistent framework for integrating different GPGPU languages to adapt Object Oriented Programming or design pattern for GPGPU based image processing has not been addressed considerably. Note that previous GPU-based image processing libraries are implemented using procedural programming, which limits the scope of code reusability, information and complexity hiding, and flexibility.

In this paper, we present an object oriented framework for GPGPU-based image processing using GLSL and CUDA. We present a class hierarchy which is based on Object Oriented Programming (OOP). The design and programming advantages of OOP paradigms enhance strongly the proposed framework in terms of code reusability / extensibility, flexibility, information hiding, and complexity hiding. We incorporate shader (GLSL) and kernel (CUDA) programming to the proposed framework. Performance is evaluated in terms of programming effort, execution overhead, and speedup factor achieved when compared to CPU.

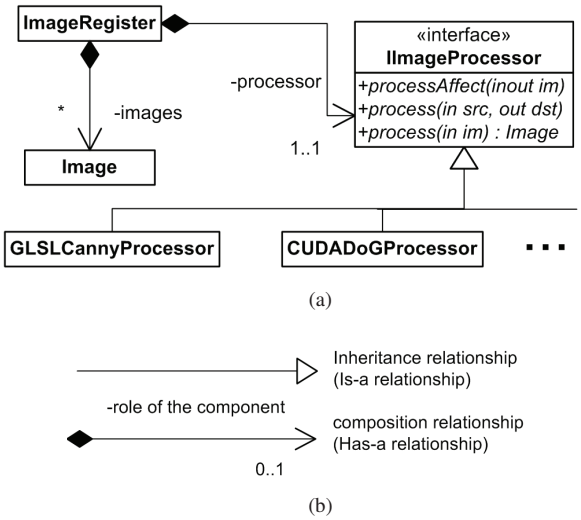


Fig. 1. General structure of the proposed framework. (a) Framework structure in UML (Unified Modeling Language); (b) Legend in UML.

2. PREVIOUS WORKS

GPU-based libraries for image processing and computer vision have been developed in GPUCV [4], and OpenVIDIA [5] projects. GPUCV provides seamless acceleration with the familiar OpenCV [6] interfaces. Recently, NVIDIA released an open source image processing library, known as NPP [7], which exploits GPU architecture for accelerating common image processing algorithms. They provide the low level API support in the development of higher level algorithms. However, they are implemented using procedural programming and lack the benefits of an object oriented framework. MinGPU [8] proposed a general purpose computation library based on object oriented framework. However, its class hierarchy is limited to be used in general image processing or computer vision. In addition, MinGPU does not support CUDA.

3. GENERAL STRUCTURE OF THE FRAMEWORK

3.1. Structure Overview

The proposed structure revolves around two central concepts: images and processes. The key idea is to separate the process from the

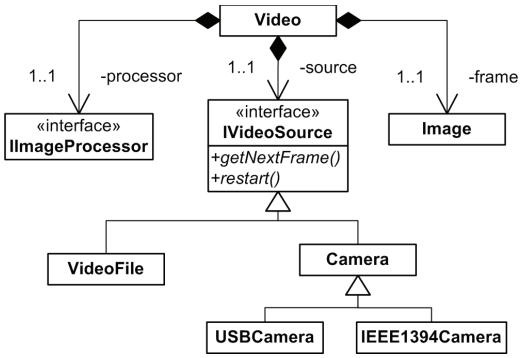
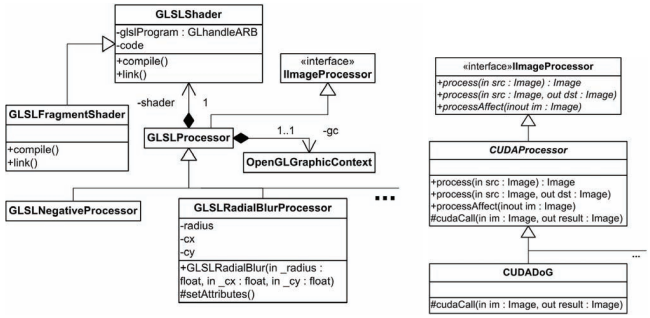


Fig. 2. The video processing structure.



(a) (b)

Fig. 3. GPGPU integration. (a) GLSL; (b) CUDA.

image, so that the target algorithm is isolated from the programming context. The extensibility of the framework is therefore improved.

The proposed framework is structured as shown in Fig. 1. At the highest level is the *ImageRegister* class, which serves as a container for image(s). The *Image* class contains all the information about image data, such as pixel color, spatial resolution, etc. Next in the hierarchy is the *IImageProcessor* interface, which provides abstraction of the available processes and receives information from the *ImageRegister* class to process the target algorithm. Each of the processing algorithms is implemented as a subclass of *IImageProcessor*.

In the proposed framework, we employ the *strategy design pattern* [9] to isolate the algorithms from the rest of the structure. The algorithm decoupling offers significant benefits. Using inheritance, a whole algorithm hierarchy can be built. Data managing routines can also be factorized. The strategy design pattern also allows interchange of algorithms that can be used independently.

3.2. Class Details

The *Image* class accommodates the data model for a 2D image with width, height, channel count, data format, and the pixel data as its parameter set. The pixel data is implemented using C++ template and supports multiple data types. The number of channels is deduced from the the image data format. The *Image* class provides functionalities such as image construction from image files, predefined pixel values and other existing images, getters and setters for all the members, and image saving on the file system.

The *IImageProcessor* interface plays the role of the abstraction to the image processing algorithms. It is the base interface to integrate any kind of image processing algorithm in the framework. The methods to implement are *processAffect*(*Image*) which overrides the original image data ; *process*(*src*, *dst*) which processes *src* and puts the result data in *dst* ; and *process*(*src*) which processes *src* and creates and returns a new image containing the result data.

The *ImageRegister* class is implemented, which is the container for multiple images. The *ImageRegister* class plays a key role in linking the *Image* class and the *IImageProcessor* interface.

3.3. Video Processing

The proposed framework provides functionalities to capture and process video frames in real time. The *Video* class is the root class in the hierarchy. It uses a single *Image* as the current frame of the *Video*. It also owns an *IImageProcessor* to process an input frame in real time. The source of the video is made abstract so that the *Video* class can independently use different video sources to capture frames. As the video source is dynamic and abstract, other type of video source can be implemented and integrated in the framework. The user has to implement methods, which are actually used to capture a video frame and set the current frame data. More specifically, the user implements the *IVideoSource* interface and *getFrame* method. Different video sources can be added to the above interface.

4. GPGPU INTEGRATION

Since the framework proposed in Fig. 1 is implemented based on the strategy design pattern, it is straightforward to create different categories and processing groups. This makes it possible to integrate different GPU interfaces in the same framework. The proposed framework is implemented to support both CUDA and GLSL.

4.1. GLSL Integration

The main class in the GLSL integration is the *GLSLProcessor* class. Individual image processing algorithms are implemented as subclasses of it. All the GLSL compilation and execution operations are implemented in the abstract class, which reduces the programming burden significantly. The GLSL integration is structured as shown in Fig. 3(a).

The *GLSLProcessor* uses the *OpenGLGraphicContext* object. All OpenGL primitive operations that are needed to perform GLSL processing are centralized in this. It includes loading images to texture memory, rendering, and reading the GPU memory.

4.2. CUDA Integration

The proposed class structure for CUDA integration is shown in Fig. 3(b). At the highest level is the *CUDAProcessor* class. As the *GLSLProcessor* class does for the GLSL algorithm integration, *CUDAProcessor* provides an integration framework for CUDA-based algorithms. The *CUDAProcessor* class incorporates the process and *processAffect* methods. The *CUDAProcessor* class also implements a *cudaCall* abstract method. This method provides the entry point for calling the CUDA kernel functions. Appropriate CUDA memory management (i.e. allocation and transfer) should also be implemented in the *cudaCall* routine. If the user wants to add a new CUDA algorithm, the task will be (i) to implement the constructor (ii) to implement the call to the appropriate CUDA function (iii) to release the texture memory (iv) to update the output image.

5. ADDITIONAL FEATURES

5.1. Feature Detection

Feature detection is a common method in computer vision in which features are extracted to analyze the contents of an image. As part of the proposed framework, we provide a programming interface for feature detection. The *ImageFeatures* is a superclass for all different feature types. This interface facilitates to integrate custom image features in the proposed framework.The *ImageFeatures* objects are generated by *IImageProcessor* objects, which are specialized in this task. These *IImageProcessor* objects apply the algorithm and add feature data to the Image.

5.2. Composite Processor

In the proposed framework, the ability to create cascading algorithms composed by multiple *IImageProcessor* objects is provided. This operation is handled by the *CompositeProcessor* class. The user can instantiate a *CompositeProcessor* object and add all the desired *IImageProcessor* objects inside, and then use it as a single *IImageProcessor* object performing multiple algorithms. This structure is implemented based on the composite design pattern as in [9].

6. EXPERIMENTAL RESULTS

To evaluate the performance and effectiveness of the proposed framework, we provide two experiment scenarios. In the first case, we select two popular image processing algorithms and evaluate their performance using CPU multicore, GPU procedural programming, and the proposed framework. We adopt CUDA and GLSL for implementing algorithms on GPU. In second case, we select a couple of use case scenarios to show the extensibility of the proposed framework.

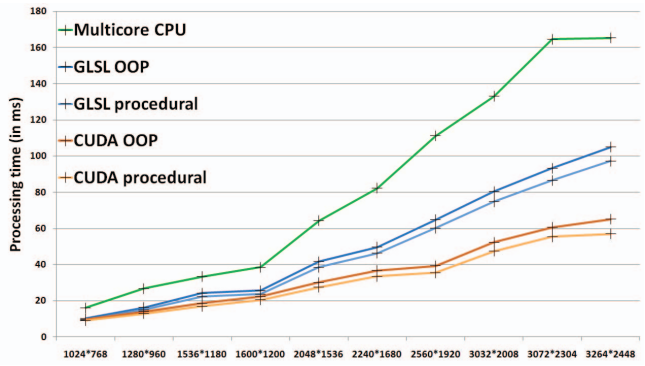
Our experimental setup equips Intel CPU (Core2 Quad Q9550 2.83GHz with 3.25 GB of RAM) and NVIDIA GPU (GTX 280 with 1GB of graphics RAM). GTX 280 has 30 multiprocessors with 240 CUDA processing cores which run at a peak performance of 933 GFLOPS while Q9550 runs at 45.28 GFLOPS.

6.1. Performance Evaluation

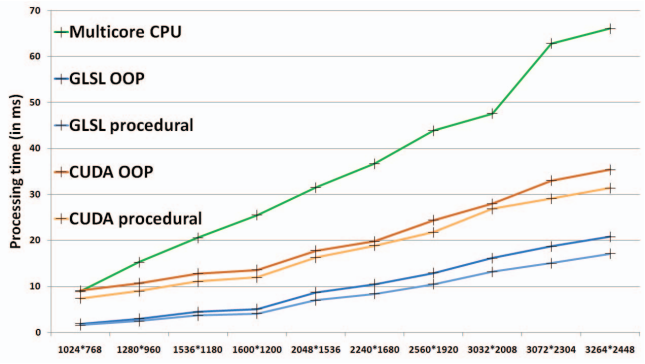
An object oriented framework typically involves additional overhead due to constant function calls in the class hierarchy and data hiding mechanism. To evaluate this overhead, we compare the performance of the proposed object oriented framework with the procedural programming case in terms of total execution time.

We select two popular algorithms, namely Canny edge detection and bilateral filtering. Both the algorithms are implemented on the GPU using the proposed framework and the stand-alone procedural programming application as well. For the CPU case, OpenCV framework is employed. The execution time is calculated as the mean of multiple iterations. Fig. 4 shows the execution time in milliseconds for varying image resolution (1∼8Mpixels).

As shown in the results, the proposed framework lags when compared to the procedural implementation, which is primarily due to the overhead in object oriented framework. However, in most cases the overhead is negligible when compared to the speedup achieved over the CPU multicore implementation. When the instruction count of the fragment shader is low, GLSL implementation achieves higher performance than CUDA implementation; whereas when the instruction count is high, CUDA implementation performs with lower processing time than GLSL.



(a)



(b)

Fig. 4. Comparison of execution time. (a) Bilateral filtering; (b) Canny edge detection.

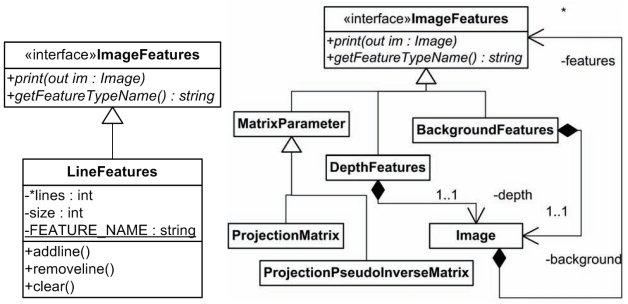
6.2. Use Case Scenarios

In order to show the extensibility of our proposed object oriented framework, we integrate two advanced algorithms, namely Nevatia and Babu's linear feature extraction and multiview stereo matching as the use case scenarios.

6.2.1. Linear Feature Extraction

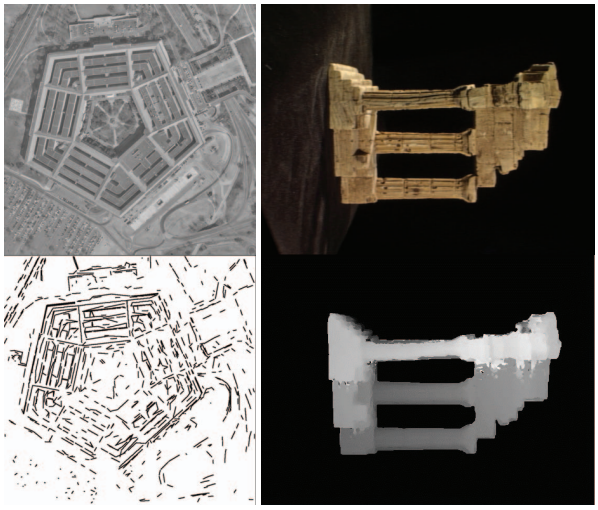
The linear feature extraction involves Canny edge detection followed by edge thinning, linking, and iterative line fitting procedures. To include linear feature extraction, the proposed framework is appended with the *LineFeatures* and the *CUDALineDetector* classes. The *LineFeatures* class contains the extracted features and implements the *ImageFeatures* interface as described in Section V. Its structure is shown in Fig. 5(a). The *CUDAFeatureDetector* is the entry point for the feature detection algorithm when using CUDA programming interface. The *CUDALineDetector* implements the linear feature extraction algorithm. An abstract *generateFeatures* method is also provided to add the extracted features to the Image. The algorithm flow for the line feature detection implementation of *generateFeatures* is as follows:

- Allocate the device memory for input and output data and upload the input data to the allocated device memory
- Call the main CUDA function which applies the algorithm
- Rearrange the data downloaded from the device memory in order to adapt to the *LineFeatures* class



(a) (b)

Fig. 5. Class structure for the *use case scenarios*. (a) Linear feature extraction; (b) Multiview stereo matching.



(a) (b)

Fig. 6. Visual results of the use case scenarios. (a) Line segment extraction of *Pentagon*; (b) Multiview stereo matching of *TempleRing*.

- Create a *LineFeatures* object
- Add the newly created object to the processed *Image* object
- Release the device (CUDA) memory

Fig. 6(a) shows the extracted line segment. The executing time in Table 1 shows that the overhead of object oriented framework is small. Note that the efficient CPU algorithm cannot be parallelized. Hence a modified algorithm is used for implementation on the GPU. Consequently, the CPU processing time in Table 1 is of the same order of magnitude when compared to GPU time.

6.2.2. *Multiview Stereo Matching*

DepthFeature, *BackgroundFeature*,*IndexFeature* and *MatrixParameter* classes are implemented following the same procedure as for the linear feature extraction example. They model parameters and features which are involved in the stereo matching algorithm. They also take advantage of the existing data structures in the framework by reusing images in this case to model depth maps and background information. *CUDADepthComputer* class is the one integrating the stereo matching in the framework. The result is shown on Fig. 6(b). The processing time in different frameworks is compared in Table 1. The performance difference between object oriented and procedural frameworks are negligible compared with the gain over CPU multicore framework.

Table 1. Processing time on different frameworks (in milliseconds).

Computing Framework	Line Feature Extraction	Multiview Stereo Matching
Multiview CPU	72	19,539
GPU in Procedural	49	185
GPU in Object Oriented	56	200

7. CONCLUSION

In this paper, we proposed an image processing framework on GPU based on object oriented paradigms and design patterns. The developed frame showed increased flexibility and straightforwardness towards GPGPU-based algorithm implementation. It is shown that the code developed using our framework can be easily reused at many levels and in different contexts. Experimental results showed that the overhead over procedural implementation is negligible when compared to the speedup achieved over the CPU multicore implementation.

8. ACKNOWLEDGEMENT

This research was supported by the MKE (The Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the NIPA (National IT Industry Promotion Agency) (NIPA-2010-(C1090-1011-0003)).

This work was supported by Mid-career Researcher Program through NRF grant funded by the MEST (No. 2009-0083945).

9. REFERENCES

[1] *General Purpose GPU Programming (GPGPU) Website*, http://www.gpgpu.org.

[2] R.J. Rost, *OpenGL Shading Language*, Addison-Wesley, 2006.

[3] D.B. Kirk and W.W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kauffmann, 2010.

[4] Y. Allusse, P. Horain, A. Agarwal, and C. Saipriyadarshan, "GpuCV: An opensource gpu-accelerated framework for image processing and computer vision," in *Proc. of the 16th ACM international conference on Multimedia*, October 2008, pp. 1089–1092.

[5] J. Fung, S. Mann, and C. Aimone, "OpenVIDIA: Parallel gpu computer vision," in *Proc. of the 13th annual ACM international conference on Multimedia*, November 2005, pp. 849–852.

[6] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*, O'Reilly, 2008.

[7] *NVIDIA NPP Library*, http://www.nvidia.com/object/npp.html.

[8] P. Babenko and M. Shah, "MinGPU: A minimum gpu library for computer vision," *Real-Time Image Processing*, vol. 3, no. 4, pp. 255–268, 2008.

[9] E. Gamma, R. Helm, R. Johnson, and J.M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.