IMPLEMENTATION AND OPTIMIZATION OF IMAGE PROCESSING ALGORITHMS ON HANDHELD GPU

Nitin Singhal[†]*, In Kyu Park*[‡]*, and Sungdae Cho*[†]

[†] Digital Media & Communication R&D Center, SAMSUNG Electronics Co. Ltd., Suwon, Korea.
[‡] School of Information and Communication Engineering, Inha University, Incheon, Korea.

ABSTRACT

The advent of GPUs with programmable shaders on handheld devices has motivated embedded application developers to utilize GPU to offload computationally intensive tasks and relieve the burden from embedded CPU. In this work, we propose an image processing toolkit on handheld GPU with programmable shaders using OpenGL ES 2.0 API. By using the image processing toolkit, we show that a range of image processing algorithms map readily to handheld GPU. We employ real-time video scaling, cartoon-style non-photorealistic rendering, and Harris corner detector as our example applications. In addition, we propose techniques to achieve increased performance with optimized shader design and efficient sharing of GPU workload between vertex and fragment shaders. Performance is evaluated in terms of frames per second at varying video stream resolution.

Index Terms— GPU, GPGPU, mobile computing, OpenGL ES 2.0, mobile devices

1. INTRODUCTION

The mobile phone is continuing to revolutionize our everyday lives. It has transformed from a simple communicator to a personal multifunction, multimedia device. The modern mobile phone is a visual computing powerhouse. It has a capable CPU, high quality color display, and co-processors or DSPs for image/video encoding and decoding. In addition, the imaging technology has changed significantly over the past few years. Today camera phones with $3 \sim 5$ mega pixels and with video capture capability are quite common. With the seemingly un-wavering boom in sales of these multimedia devices, the opportunity to develop and sell sophisticated mobile applications is ever more appealing. Software is emerging as the major linchpin in the battle for dominance in this fast growing mobile application sector. However, there are many challenges facing application developers wishing to target mobile phones. Compare to the PC platform, the mobile phone platform is limited by (i) power supply; (ii) computational power; (iii) physical display size; and (iv) input modalities. In addition, modern mobile phone processors lack a floating point unit (FPU). This makes using integer or fixed-point arithmetic obligatory, which reduces the accuracy.

The graphics processing unit (GPU), which originally was used exclusively for visualization purposes, has evolved into an extremely powerful co-processor. Efforts in General Purpose Computation on GPU (GPGPU) [1], research have created a wealth of opportunities for PC software developers to offload computationally intensive tasks to the GPU. The advent of GPUs with programmable shaders on handheld devices have motivated application developers to utilize GPU for tasks other than graphics rendering and find ways to use this co-processor to relieve the burden from embedded CPU.

The contribution of this paper is two-fold. First, we propose an image processing toolkit on handheld GPU with programmable shaders using OpenGL ES 2.0 API [2]. The idea is to upload the dataset into the graphics hardware as texture and perform all possible tasks on the GPU. We show that a range of image processing algorithms map readily to handheld GPU by using real-time video scaling, cartoon-style non-photorealistic rendering (NPR), and Harris corner detector as our example applications. The second contribution of this paper is to propose techniques to achieve increased performance with optimized shader design and efficient sharing of GPU workload between vertex and fragment shaders. Performance is evaluated in terms of frames per second (fps) at varying video stream resolution.

2. RELATED WORK

On the PC platform, through development of elaborate interfaces such as CUDA (Compute Unified Device Architecture) [3] and OpenCL [4], GPU can be used to process data and deal with computationally intensive tasks. These interfaces increases user programmability and facilitates use of GPU for general purpose. A survey on GPGPU is intensively described in [5].

Image processing has gained considerable attention among GPGPU researchers. Most image processing operations perform the same computation on a number of pixels; thus they can exploit the SIMD (single instruction multiple data) architecture and be effectively implemented on the GPU. Recently, GPUCV [6], MinGPU [7], and OpenVIDIA [8] have emerged as open source image processing and computer vision libraries based on GPGPU technique. However, they are mainly targeted on the PC platform using interfaces such as CUDA [3], which are not available on the newest generation of handheld GPU.

3. OPENGL ES 2.0 GRAPHICS PIPELINE

The OpenGL ES is a graphics APIs standard for the embedded systems [2]. Fig. 1 shows the OpenGL ES 2.0 graphics pipeline. The shaded boxes in Fig. 1 indicate the programmable stages of the pipeline. The vertex shader implements a general-purpose programmable technique for operating on the vertices. Vertex shaders can be used for traditional vertex-based operations such as transforming the position by a matrix, computing the lighting, and generating or transforming texture coordinates. In the primitive assembly stage, the shaded vertices are assembled into individual geometric primitives that can be drawn such as a triangle, line, or point-sprite. The next stage is the rasterization phase. Rasterization is the process that converts primitives into a set of two-dimensional fragments, which are processed by the fragment shader. These two-dimensional fragments represent pixels that can be drawn on the screen. The fragment

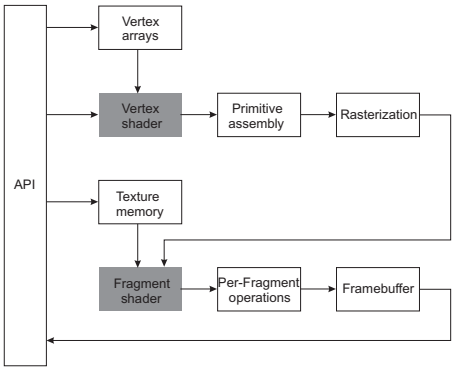

Fig. 1. OpenGL ES 2.0 Graphics Pipeline

shader process the per-pixel algorithm and generates a color value for each fragment. At the end of the per-fragment stage either the fragment is rejected or a fragment color is written to the screen.

4. IMAGE PROCESSING TOOLKIT ON HANDHELD GPU

Image processing algorithms typically involve independent processing of massive pixels or feature set, which can benefit from SIMD style GPU architecture. This fact makes image processing a prime topic for porting to the GPU.

We provide the following functionalities as part of the image processing toolkit.

- *Sampling*: Upscale/downscale
- *Color transformation*: RGB2GRAY, RGB2YCbCr, YCbCr2-RGB, RGB2HSV, HSV2RGB
- *Convolution operations*: Gaussian blur, Sharpening, Gradient, Bilateral filter, Laplacian filter, Box filter, User-defined kernel
- *Segmentation*: Bloom effect (thresholding), Skin detection
- *Edge detection*: Sobel, Prewitt
- *Image Enhancement*: Detail Enhancement, Contrast stretching, Edge enhancement, Median filtering
- *Effects*: Sepia, Radial blur, Negative, Color gradient, Edge overlay, Gray, Gamma, Edge
- *Comparison-based operations*: Dilation, Erosion, Median, Zero-crossing

Each of the above image processing operation is implemented on the GPU using one or more rendering cycles. The rendering procedure typically involves the following steps:

- *Initialization*: Compile and load the shaders to the GPU memory, as well as allocate necessary Framebuffer objects and texture memory;
- *Render Scene*: Fetch a frame from the video stream and bind the frame to the texture memory. Invoke the fragment program. This is done by passing the vertices to the GPU. Typically, a quadrilateral oriented parallel to the image plane, sized to cover a rectangular region of pixels;
- *Graphics pipeline*: The vertex shader transforms the vertices from world coordinates to image coordinates. The rasterizer generates a fragment for every pixel location in the quadrilateral. Each of the generated fragment is then processed by the fragment shader program;

Table 1. The number of arithmetic instructions and rendering cycles used for different operations when OpenGL ES shading language code is compiled

Operation	Number of instructions	Number of rendering cycles	Operation	Number of instructions	Number of rendering cycles
RGB2GRAY	5	1	Gaussian	21	1
RGB2YCbCr	14	1	Sharpening	13	1
YCbCr2RGB	14	1	Gradient	19	2
RGB2HSV	28	1	Bilateral	62	2
HSV2RGB	29	1	Laplacian	14	1
			Box filter	18	1
Bloom	15	1	Sobel	24	2
Skin detection	25	2	Prewitt	16	2
Detail enhancement	13	1	Contrast Stretching	13	1
Edge enhancement	25	2	Median filtering	43	1
Dilation	22	1	Erosion	22	1
Median	43	1	Zero-crossing	22	1
Sepia	21	1	Color gradient	20	1
Radial Blur	21	1	Negative	2	1
Edge overlay	25	2	Gamma	15	1
Gray	5	1	Edge	24	2

- The output of the fragment program is a value (or color) per fragment.

In a typical code translation from CPU to OpenGL ES 2.0 graphics pipeline, the vertex shader along with the rasterizer is used to generate fragment coordinate corresponding to each pixel location. The fragment shader is then used to process the per-pixel algorithm, which is common for all the pixels.

In mobile devices, the battery life is limited. The power consumption of modern mobile phones is two orders magnitude lower than PC graphics cards. As a consequence, the number of instructions slots for a vertex respectively fragment shader is limited by the handheld GPU architecture. The vertex shader is rarely a bottleneck considering GPUs excellent vertex processing capability. However, the fragment shader with a large number of instructions is largely expected to be a bottleneck when applied to large number of pixels. The number of instructions determines the performance (framerate) achieved. Lower the instruction count, better the performance would be. Table 1 shows the number of instructions used for different image processing operations. If a certain operation does not fit into one rendering pass, the intermediate result can be rendered into pbuffer. A pbuffer is an offline rendering context which is not displayed, but has the same feature as a screen framebuffer. Pbuffers can be directly bound as texture in the next rendering pass to be processed by the fragment program. Table 1 also shows number of rendering cycles required for each operation.

5. PERFORMANCE OPTIMIZATION

5.1. Floating Point Precision Control

To achieve high throughput, OpenGL ES 2.0 supports three precision modifiers (i) *lowp*; (ii) *mediump*; (iii) *highp*. The *highp* precision is represented as 32 bit floating point variables, the *mediump* modifier is represented as 16 bit floating point values, covering the range [-65520 65520], and the *lowp* precision use a 10 bit fixed point format, allowing values in the range [-2 2), with a precision of 1/256. The *lowp* precision is useful for representing colors and any data read from low precision textures. Choosing a lower precision can increase performance but also introduces precision artifacts.

Fig. 2 shows three different versions of the fragment shader implementing sharpening filter. We begin with a simple version having *highp* modifier for every variable. This code uses 34 instructions per fragment processing. Performance for this code is 13 fps. By examining the code, we find that any color read from low precision textures can be represented using *lowp* precision modifier. Using *highp* precision for color read results in waste of limited set of in-

```
#define NUM_TEX 5
uniform sampler2D sTex;
varying mediump vec2 texC[NUM_TEX];
void main ()
{
    highp vec3 pCU = (texture2D (sTex, texC[1])).xyz;
    highp vec3 pLC = (texture2D (sTex, texC[2])).xyz;
    highp vec3 pCC = (texture2D (sTex, texC[0])).xyz;
    highp vec3 pRC = (texture2D (sTex, texC[3])).xyz;
    highp vec3 pCD = (texture2D (sTex, texC[4])).xyz;
    highp vec3 sum = -pCU - pLC + 5.0 * pCC - pRC - pCD;
    gl_FragColor = vec4 (sum, 1.0);
}
```

(a)

```
#define NUM_TEX 5
uniform sampler2D sTex;
varying mediump vec2 texC[NUM_TEX];
void main ()
{
    lowp vec3 pCU = (texture2D (sTex, texC[1])).xyz;
    lowp vec3 pLC = (texture2D (sTex, texC[2])).xyz;
    lowp vec3 pCC = (texture2D (sTex, texC[0])).xyz;
    lowp vec3 pRC = (texture2D (sTex, texC[3])).xyz;
    lowp vec3 pCD = (texture2D (sTex, texC[4])).xyz;
    lowp vec3 sum = -pCU - pLC + 5.0 * pCC - pRC – pCD;
    gl_FragColor = vec4 (sum, 1.0);
}
```

(b)

```
#define NUM_TEX 5
uniform sampler2D sTex;
varying mediump vec2 texC[NUM_TEX];
void main ()
{
    lowp vec3 pCC = (texture2D (sTex, texC[0])).xyz;
    lowp vec3 pCU = (texture2D (sTex, texC[1])).xyz;
    lowp vec3 pLC = (texture2D (sTex, texC[2])).xyz;
    lowp vec3 pRC = (texture2D (sTex, texC[3])).xyz;
    lowp vec3 pCD = (texture2D (sTex, texC[4])).xyz;
    lowp vec3 sum1 = 2.0 * pCC - pCU - pLC;
    lowp vec3 sum2 = 2.0 * pCC - pRC - pCD;
    gl_FragColor = vec4 (sum1 + sum2 +pCC, 1.0);
}
```

(c)

Fig. 2. Sharpening filter fragment shader design. (a) version 1; (b) version 2; (c) version 3

structions. Fig. 2(b) shows the code with *lowp* precision for each variable. This code uses fewer instructions per fragment (15 in this case). However, multiplying a *lowp* precision vector pCC with 5.0 results in overflow from the numeric range of *lowp* precision. This causes the intermediate value to be clamped within [-2 2), resulting in incorrect sum value. Fig. 2(c) shows the optimized version using *lowp* precision for color read and with careful intermediate computations. All intermediate values $2.0 * pCC, sum1$, and $sum2$ are within [-2, 2), the numerical range of *lowp* precision variables. This code evade any overflow and uses 13 instructions per fragment with a performance of 29 fps. Note that in OpenGL ES 2.0, all color read from global texture memory are normalized to [0,1].

5.2. Load Sharing Between Vertex And Fragment Shaders

Convolution operations such as filtering require accessing neighborhood pixels to compute fragment output. To obtain the value of a particular neighborhood pixel that contributes to the filtering output value, normally it would be necessary to compute the addresses in a fragment shader. For a 3×3 filter size, this would require 15 additional instructions to compute the addresses. For illustration, consider the example of 3×3 Sobel edge detection operation. Computing the neighborhood texture addresses in a fragment shader requires 39 instructions and achieve a throughput of 13 fps at VGA (640×480) stream resolution.

For general image processing operations, the number of vertices processed is much lower than the total number of fragments, which are millions in number. Consequently, operations per vertex are significantly cheaper than per fragment and it is generally recommended to perform calculations per vertex instead of per fragment, whenever possible. In case of filtering, the straightforward way is to pre-compute neighboring texture addresses in a vertex shader. This would reduce the fragment shader instruction count and leads to increased performance. For example, the same Sobel edge detection fragment shader requires 24 instructions with load sharing with vertex shader and achieves a performance of 29 fps. Output from the vertex shader is represented by *varying* modifier, which is first interpolated by the rasterizer and then fed into the fragment shader. Modern handheld GPU architecture supports up to 8 *varying* vectors between vertex and fragment shaders. Each *varying* vector is a four-dimensional vector, typically ordered with xyzw notation. In case of 2D texture coordinate, xy components are used for storing one address. The flexibility of the hardware allows to use zw components for storing address as well. Hence, a vertex shader can output up to 16 texture coordinates.

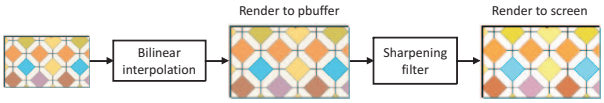

Fig. 3. Real-time Video Scaling

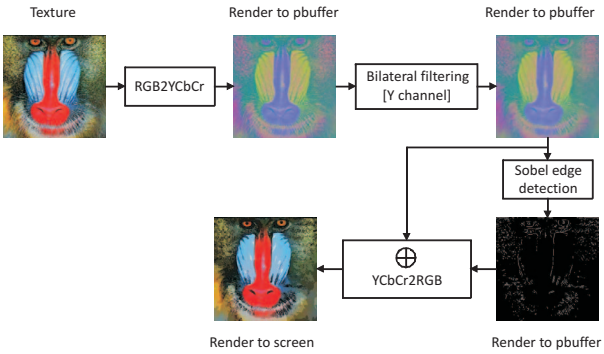

Fig. 4. Cartoon-style Non-Photorealistic Rendering

6. APPLICATION SUITE

We performed an application study with the intent of testing the applicability of the image processing toolkit on real applications. We have selected a suite of applications comprising of real-time video scaling, cartoon-style NPR, and Harris corner detector.

6.1. Real-time Video Scaling

Scaling for real-time video stream has more constraints than for still images. One major constraint is the need to process each incoming frame in a given amount of time. A well established method for video scaling is using bilinear interpolation. However, it suffer from interpolation artifacts. To improve the subjective quality of the bilinear interpolated rendering output, we post-process the interpolated frame using a 3×3 sharpening filter. Fig. 3 shows the block framework. The framework involves two rendering cycles, one to obtain the interpolated frame using bilinear interpolation, and other for sharpening filter.

6.2. Cartoon-style Non-Photorealistic Rendering

In this work, we present an automatic, real-time video and image abstraction framework that abstracts images by modifying the contrast of visually appealing features. Fig. 4 shows the basic framework

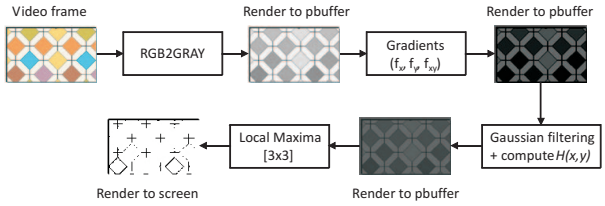

Fig. 5. Harris Corner Detector

Table 2. Application suite performance in frames per second (fps)

Application \ Resolution	320×240 (fps)	400×300 (fps)	480×360 (fps)	560×420 (fps)	640×480 (fps)
Real-time video scaling	60	54	42	35	28
Cartoon-style NPR	19.7	15.3	11.8	9.4	6.7
Harris corner detector	19.4	15	11.4	8.8	6.5

of our workflow. Given the input image $f(x, y)$, this image is convolved by a Bilateral filter kernel. Then, the highlighting edges are added to increase local contrast and sharpen the resulting cartoon-style image.

6.3. Harris Corner Detector

Corners are local image features characterized by locations where variation of intensity function $f(x, y)$ in both X and Y directions is high, i.e., both first order derivatives f_x and f_y are large. General corner detector algorithms are based on the local structure matrix

$$C = w(x, y; \sigma) * \begin{pmatrix} f_x^2 & f_x f_y \\ f_x f_y & f_y^2 \end{pmatrix}. \quad (1)$$

Harris defined a measure of corner strength as

$$H(x, y) = det(C) - \alpha \cdot (trace(C))^2, \quad (2)$$

where α is constant and $H \geq 0$ if $0 \leq \alpha \leq 0.25$. A corner is detected where $H(x, y)$ achieves local maxima. Fig. 5 illustrates the four rendering cycles involved in obtaining Harris corner points.

7. EXPERIMENTAL RESULTS

In this section, we present the results of video stream analysis with GPU implementation of applications described in Section 6. We evaluate the performance in terms of frames per second achieved at varying stream resolution. We have tested the performance of our application suite on the OMAP ZOOM Mobile Development Kit (MDK). OMAP ZOOM is a compact hardware platform based on the Texas Instruments OMAP3430 SOC equipped with POWERVR SGX 530 [9] GPU, which enables support for OpenGL ES 2.0 API.

In Table 2, we summarize the frame rates for real-time video scaling, cartoon-style NPR, and Harris corner detector at varying video stream resolution. The actual shader performance on POWERVR SGX is linked to stream resolution and arithmetic instruction count. As shown in Table 2, the performance shrinks with the increase in stream resolution. Real-time video scaling involves the least amount of fragment shader instructions, achieving a performance of full 60 fps at QVGA (320×240) resolution. Fragment shader with a large number of instructions serves as a bottleneck when applied to large number of pixels as in case of cartoon-style NPR and Harris corner detector. Furthermore, the shader performance scales linearly with the clock speed. The POWERVR SGX 530 employs sophisticated power saving techniques to reduce power consumption of mobile phones (100 $\sim$ 110 mA), which limits the clock speed on SGX to 100 $\sim$ 166 MHz. The low clock speed associated with POWERVR SGX 530 GPU architecture significantly reduce the performance. Fig. 6 shows an example of cartoon-style NPR on OMAP ZOOM MDK.

Fig. 6. Cartoon-style NPR on OMAP ZOOM MDK. (a) VGA (640× 480); (b) QVGA (320 × 240)

8. CONCLUSION

In this paper, we explored the design and implementation of image processing algorithms on handheld GPU architecture using OpenGL ES 2.0 API. By using the proposed image processing toolkit, we show that a range of algorithms map readily to handheld GPU architecture using real-time video scaling, cartoon-style NPR, and Harris corner detector as our example applications. We proposed techniques for increasing performance using optimized shader design and load sharing between vertex and fragment shaders. Performance achieved is analyzed on video stream at varying resolution.

9. REFERENCES

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

[2] Khronos Group, *OpenGL ES 2.0 Specification*, http://www.khronos.org/opengles.

[3] NVIDIA Corporation, *Compute Unified Device Architecture (CUDA)*, http://developer.nvidia.com/object/cuda.html.

[4] Khronos Group, *Open Computing Language*, http://www.khronos.org/opencl/.

[5] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[6] 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.

[7] 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.

[8] 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.

[9] IMAGINATION Technologies, *POWERVR SGX*, http://www.imgtec.com.