PAPER

# Implementation and Optimization of Image Processing Algorithms on Embedded GPU

**Nitin SINGHAL**[†a)], **Jin Woo YOO**[††], **Ho Yeol CHOI**[††], ***Nonmembers***, ***and*** **In Kyu PARK**[††b)], ***Member***

**SUMMARY** In this paper, we analyze the key factors underlying the implementation, evaluation, and optimization of image processing and computer vision algorithms on embedded GPU using OpenGL ES 2.0 shader model. First, we present the characteristics of the embedded GPU and its inherent advantage when compared to embedded CPU. Additionally, we propose techniques to achieve increased performance with optimized shader design. To show the effectiveness of the proposed techniques, we employ cartoon-style non-photorealistic rendering (NPR), speeded-up robust feature (SURF) detection, and stereo matching as our example algorithms. Performance is evaluated in terms of the execution time and speed-up achieved in comparison with the implementation on embedded CPU.

***key words:*** *embedded GPU, GPGPU, image processing, OpenGL ES 2.0, NPR, SURF, stereo matching*

## 1. Introduction

The mobile phone continues to revolutionize our everyday lives. It has transformed from a simple communicator to a personal multifunction, multimedia device. The modern mobile phone is also a visual computing powerhouse. It has a capable CPU, high quality color display, co-processors or DSPs for image/video encoding and decoding, and sensors such as camera, gyroscope, and others. In particular, imaging technology has changed significantly over the past few years. Today, camera phones with 3 ∼ 5 mega pixels and with HD video capture capability are quite common. With the seemingly un-wavering boom in sales of these multimedia devices and availability of additional hardware components, the opportunity to develop and sell sophisticated mobile applications is ever more appealing.

However, there still exist many challenges facing application developers wishing to target mobile phones. Compared 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. The mobile phone is powered by batteries and it is obligatory for the system to use as little energy as possible. The power consumption is an increasing function of the clock frequency, and hence it is kept rather low. Although strides are being made to improve the clock frequency using sophisticated power-reduction techniques, the fastest CPU runs at around 1.0 GHz at most. A related problem is the limited amount of RAM, which is usually only a few megabytes. In addition, embedded processors lack a floating point unit (FPU). This makes using integer or fixed-point arithmetic requisite, which reduces the accuracy significantly.

The embedded graphics processing unit (GPU) has evolved into an extremely powerful co-processor. Over the last decade, GPU has evolved into a general-purpose programmable architecture. It now supports programming environment that makes it possible to use GPU for a wide range of non-graphics tasks, including many applications in image processing and computer vision. Efforts in general purpose computation on GPU (GPGPU) [1] research have created a wealth of opportunities for developers to offload computationally intensive tasks to the GPU. Recently, an increasing number of mobile phones are equipped with a GPU. The advent of GPUs with programmable shaders on mobile phones finds ways to use this co-processor to relieve the burden from embedded CPU. Modern embedded GPUs provide programmable vertex and pixel shaders that can be used to speed-up image processing and computer vision algorithms.

General image processing and computer vision algorithms process large data sets with complex mathematical and logical operations. These algorithms perform the same computation on a number of pixels or fragments, a typical form of data parallelism, which fits perfectly with the GPUs single instruction multiple date (SIMD) architecture and facilitates significant acceleration. However, a large number of image processing algorithms fail to achieve acceleration due to limitations of GPU architecture. In addition, embedded GPUs have hardware limitation that must be taken into account. Consequently, it is critical to analyze the algorithms for efficient parallelization on GPUs.

In this paper, we analyze the key factors underlying the implementation, evaluation, and optimization of image processing and computer vision algorithms on embedded GPU using OpenGL ES 2.0 shading language. First, we present the characteristics of the embedded GPU and its inherent advantage in processing image processing and computer vision algorithms over embedded CPU.

Next, we propose techniques to achieve increased performance with optimized shader design. To show the effectiveness of the proposed techniques and validate our approach, we employ cartoon-style non-photorealistic rendering (NPR), speeded-up robust feature (SURF) detection, and

Manuscript received August 5, 2011.
Manuscript revised December 20, 2011.

[†]The author is with the Digital Media & Communication R&D Center, Samsung Electronics Co. Ltd., Suwon 443–742, Korea.
[††]The authors are with the School of Information and Communication Engineering, Inha University, Incheon 402–751, Korea.
a) E-mail: n.singhal@samsung.com
b) E-mail: pik@inha.ac.kr

DOI: 10.1587/transinf.E95.D.1475

stereo matching as our example algorithms. To the best of our knowledge, this is the first work that uses embedded GPU for GPGPU research. An early version of this paper has been presented in a conference [2].

The remainder of this paper is as follows. Section 2 reviews notable GPGPU related research activities. Section 3 addresses the embedded GPU architecture. In Sect. 4, methods to characterize embedded GPUs are proposed. Section 5 describes the proposed techniques for performance boost. Section 6 describes the GPU design and implementation of the algorithms investigated. Experimental results are shown in Sect. 7.

## 2. Related Work

On the PC platform, through the development of elaborate interfaces such as GLSL [3], CUDA [4], and OpenCL [5], GPU can be used to process data in a massive parallel way and deal with computationally intensive tasks. These interfaces increase the user programmability and facilitate the use of GPU for general purpose. An intensive survey on GPGPU is described in [6].

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. Several image processing algorithms have been implemented on the GPU, including basic operations, such as the fast Fourier transform, convolution, differential equation-based algorithms, video encoding/decoding [7], and pattern recognition and computer vision algorithms.

GPU-based libraries of image processing and computer vision have been developed in GpuCV [8], MinGPU [9], and OpenVIDIA [10] projects. OpenVIDIA provides a framework for video input, display, and GPU processing, as well as implementations of feature detection and tracking, skin tone tracking, and projective panoramas. GpuCV is designed to provide seamless acceleration with the familiar OpenCV interfaces. Recently, NVIDIA released an open source image processing library, known as NPP [11], which exploits GPU architecture for accelerating common image processing algorithms. These libraries provide the low level API support and aid in the development of higher level algorithms. However, they are mainly targeted on the PC platform using interfaces such as CUDA, which are not available on the newest generation of handheld GPUs.

Similar to our work is that of Park et al. [12]. They analyzed general multi-core GPU on the PC platform from an image processing point of view. The main difference between our work and theirs is that we analyze embedded GPU which is largely limited in hardware capabilities and very far from PC graphics cards in terms of the degree of parallelism. Additionally, we propose techniques for optimizing fragment shader program with the focus on image processing and computer vision algorithms.

## 3. Embedded GPU Architecture

The embedded GPU architecture is built around the need to access data efficiently and schedule parallel computations. Extensive use is made of single-instruction multiple-data (SIMD) parallelism, in which one instruction causes a single operation to take place on more than one value at the same time.

The key step in the evolution of graphics hardware was replacing the fixed-function vertex and fragment operations with user-specified programs, also known as shaders. Shaders give developers a huge amount of flexibility to create complex visual effects and to offload computationally intensive tasks to the embedded GPU. The shading language supports complex data types and a rich set of control-flow constructs.

An OpenGL ES 2.0 compliant embedded GPU is a single core (or multi-core) processor, capable of executing multiple threads concurrently. This processor operates as a co-processor to the CPU and is designed on a single System-on-Chip (SoC), popularly known as an Application Processor (AP). An embedded GPU core executes multiple threads but all threads run the same set of instruction, operating on different data.

The SGX 530/540 GPU from Imagination Technologies [13] has a multithreaded architecture that processes several tasks (instructions) in parallel. A single SGX core has 16 threads, 4 of which are active at any time. However, the multithreading is internal and not visible to the user, in contrast to GPU on the PC platform. Furthermore, the vertex and fragment (pixel) processing is parallelize, i.e., when processing pixels for the current frame, the vertices for the next frame can be processed. The hardware scheduler properly manages the vertex and pixel instruction processing. In concept, this is termed as deferred rendering architecture. In most SoC designs using SGX core, the memory (LPDDR1/LPDDR2) is shared between the CPU and GPU. In addition, the GPU has a device virtual address space of 128∼256 MB, which depends on the device and the driver implementation. All the elements required for rendering are stored in this virtual address space. The SGX series GPUs are generally clocked at 100∼400 MHz and supports 16 bit and 32 bit floating-point unit.

In general, an embedded GPU is designed as a dedicated hardware for fast 3D rendering with lower power consumption. The key to good graphics performance and low power consumption is hardware design, which includes characteristics such as unified shader design, texture compression, and tiling architecture [14]. However, there exist a few key differences between desktop GPUs and embedded GPUs. Firstly, the interconnection between GPU and CPU is very narrow in embedded GPUs. This results in low memory bandwidth and heavier overhead in data transfer between the CPU and GPU. Second, embedded GPUs are designed for low power consumption, which causes lower clock speed and fewer shader units with slower speed.

Lower clock speed also extends to the speed of video memory and consequently dedicated graphics memory is often unavailable. Thirdly, embedded GPUs lack the generality of programming interfaces. We still have to program the shaders with the OpenGL Shading Language, which limits the possibility of efficient parallelization of general-purpose problems. Finally, embedded GPUs have few (currently less than four) cores in comparison to many-core (more than hundreds) desktop GPU, which exhibits significant limitation in handling problems with high computational complexity.

## 4. Image Processing on Embedded GPU

In practice, image processing and computer vision algorithms involves intensive floating point and logical operations. These operations are independently processed using the SIMD-style multithreaded GPU architecture. Furthermore, image processing algorithms involves large memory buffers and needs frequent access to them. In this section, we present characteristics of an embedded GPU, with the focus on parallel implementation of image processing and computer vision algorithms.

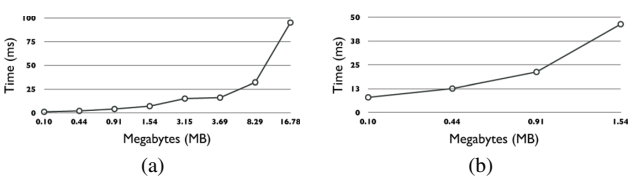
### 4.1 Memory Transfer Bandwidth

Textures provide the detail required to present images for rendering. The time required to transfer image buffer from CPU to GPU texture and vice versa, is crucial to any parallel implementation on embedded GPU. In mobile phone application processor (SoC), the memory is shared between CPU and GPU. However, textures have to be properly wrapped for the graphics core and cannot be accessed directly as the CPU image array. The memory transfer overhead results in significant bottleneck for algorithms involving large memory buffers and low floating point computations. Algorithms involving high floating point intensity can successfully hide the memory transfer latency. Figure 1 shows the memory transfer bandwidth on the mobile phone with POWERVR SGX 540 GPU (200 MHz) and ARM CORTEX A8 CPU (1 GHz). For the above CPU/GPU configuration, the memory bandwidth stands at 220.54 MB/sec (CPU to GPU) and 30.92 MB/sec (GPU to CPU).

### 4.2 Floating Point Vs. Fixed Point Implementation

Embedded GPU is designed such that more transistors are allocated to data processing. More specifically, it is well suited to algorithms with high floating point intensity. Furthermore, embedded GPUs outperforms their CPU counterparts in floating point operations per second. Embedded CPUs either lack a floating point unit (FPU) or relies on software math libraries, which prohibitively slow down the floating point implementation. Here we compare the ARM CORTEX A8 CPU with the POWERVR SGX 540 GPU in terms of the floating/fixed point design. The SGX 540 GPU has fast vectored floats compared to the slow VFP-Lite hardware acceleration of CORTEX A8. Furthermore, ARM's Neon hardware is not fully optimized for floating point arithmetic.

Considering the above context, an image processing algorithm that has many floating point operations is likely to have higher speedup when implemented on embedded GPU. For example, a 5×5 Gaussian filter, when implemented on an embedded CPU with floating point design (3074.8 ms) lags behind the fixed point implementation (207.1 ms) by a significant proportion. Also, the parallel implementation on the GPU (48.90) outperforms the CPU fixed point implementation.




(a) (b)

**Fig. 1** Memory transfer rate. (a) CPU to GPU. (b) GPU to CPU.

### 4.3 Shader Instruction Count Vs. Number of Rendering Cycles

The most important criterion when designing image processing algorithms on the mobile GPU is the balance between fragment shader instruction count and number of rendering cycles.

In mobile devices, the battery life is limited. The application processor (SoC), is designed to consume low power. As a result, the number of instruction slots for a vertex respectively, fragment shader is limited. The vertex shader is rarely a bottleneck considering GPUs excellent vertex processing capabilities. However, the fragment shader with large number of instructions is likely to become a bottleneck when applied to a large number of pixels. For general image processing operations, the number of vertices processed is much lower than the total number of fragments. As a result, the number of fragment shader instructions determines the performance achieved. Lower the instruction count, better the performance. On the other hand, in a typical parallel implementation, the problem at hand is partitioned into subproblems (rendering cycles) that are executed sequentially. However, having multiple rendering cycles reduces the parallel fraction of the entire problem, which significantly impact the maximum speed-up achieved.

Considering the above context, there is a trade-off between combining multiple rendering cycles in a single pass and splitting a single pass into multiple rendering cycles. Packing multiple rendering cycles into a single fragment shader increases the instruction count. On the other hand, increasing the number of rendering cycles reduces the parallel fraction.

## 5. Performance Optimization

The performance of OpenGL ES 2.0 applications differs from that of OpenGL on desktop operating system. Embedded GPUs are optimized for low memory and power usage, using techniques different from a typical desktop GPU. Designing shaders inefficiently not only results in poor frame rate, but also significantly reduces the battery life.

In this section, we present techniques to optimize the shader performance. The techniques are customized for image processing context and address the need for compact shader code that match the smaller hardware limits of the embedded GPU.

### 5.1 Floating Point Precision Control

Precision hints were added to the OpenGL ES shading language specification to address the need for compact shader variables that match the limited hardware capabilities of embedded devices. Shader variables use precision to provide hints to the compiler on how the variable is used in the application. OpenGL ES 2.0 supports three precision modifiers (i) *lowp*; (ii) *mediump*; (iii) *highp*. The *highp* precision variable is interpreted as a single precision, 32 bit floating point value. The *mediump* precision variable is interpreted as a half-precision floating point value (16 bit), covering the range $[-65520, 65520]$. Lastly, the *lowp* precision variable is interpreted with a 10 bit fixed point format, allowing values in the range $[-2.0, 2.0)$ with a precision of 1/256. Generally, when in doubt, *highp* precision is used as default. The *lowp* precision is useful for representing colors in the 0.0 to 1.0 range. Choosing a lower precision increases the performance but may introduce artifacts. After reducing the precision, it is highly recommended to retest the application to avoid any overflow occurred due to the numerical range of lower precision modifiers.

### 5.2 Loop Unrolling

OpenGL ES 2.0 offers full support for flow control operations such as **for** and **while**. However, to process a loop a shader need more instructions in increment and comparison operations. Eliminating loop by either an optimized unrolling or vector utilization to perform operations, results in lower instruction count and helps achieves higher performance.

Note that when the loop cannot be unrolled, it is preferred to have a constant loop count so that dynamic branching is reduced.

### 5.3 Branching

Branches are discouraged in shaders, as they can significantly degrade the ability to execute operations in parallel threads. Branching impacts the shader performance depending on the type of branching variable. Branching on a constant known value achieves the best performance, followed by branching on a uniform variables. Branching on a value computed inside the shader results in significantly low performance.

### 5.4 Load Sharing Between Vertex and Fragment Shaders

Many image processing and computer vision algorithms involves convolution operations such as filtering, which require accessing neighborhood fragments to compute output. To obtain the color of a particular neighborhood fragment, it is common to compute the texture coordinate inside the fragment shader. This is commonly known as dynamic texture read or dependent texture read. A dependent texture read occur when a fragment shader computes the texture coordinates rather than using the unmodified texture coordinates passed into the shader. Although OpenGL ES 2.0 shader language support this, every time a dependent texture read is encountered, a stall occurs until the texture information has been retrieved. When there are a small number of texture reads, the performance degradation from the stall is hidden. The reason is that the hardware has the ability to schedule other operations until the texture data has been retrieved. As the number of dependent texture read increases, the number of additional operations that can be performed that do not rely on dependent texture reads decreases, which results in a significant bottleneck. This is because the hardware runs out of operations and reaches a idle state, waiting for the data to be retrieved from external memory.

For 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, so it is generally recommended to perform calculations per vertex. In case of filtering, the straightforward way is to precompute neighboring texture coordinates in a vertex shader. By moving the calculations to the vertex shader and directly using the vertex shader's computed texture coordinates, the fragment shader avoids the dependent texture read.

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 embedded 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 a single coordinate. The flexibility of the hardware allows us to use **zw** components for storing coordinates as well. Although packing multiple sets of texture coordinates into a single varying parameter and using a swizzle command to extract the coordinates causes a dependent texture read, we can still avoid calculating indices in fragment shaders.

### 5.5 Texture Compression

Memory reduction is always performed at the cost of final rendered image quality. However, there are image process-

ing applications such as feature detection, edge detection, where the input image is processed without considering the final rendered image quality. Texture compression usually provides the best balance of memory savings and quality. OpenGL ES 2.0 supports the POWERVR Texture Compression (PVRTC) format. There are two levels of PVRTC compression, which offers a 8:1 and 16:1 compression ratio over the uncompressed 32-bit texture format. A compressed PVRTC texture provides a decent level of quality, particularly at the 8:1 (4-bit level) compression ratio. If the texture cannot be compressed, a lower precision pixel format such as, RGB565, RGBA5551, or RGBA4444 can be used. These lower precision formats uses half the memory of a texture in RGBA8888 format and help reduces the memory transfer time to and from the GPU.

### 5.6 Optimization Example

In this subsection, we employ 5×5 Gaussian blur filter and evaluate the impact of different optimization techniques discussed above. Figure 2 shows the basic fragment shader implementation using **for** loop. Table 1 shows the speed-up achieved after each optimization step. Firstly, the **for** loops are eliminated by unrolling the loop. The fragment shader with the unrolled loop outperforms the shader with the **for** loop with significant speedup (5x). Next, we optimize the unrolled loop shader using load sharing. As discussed above, embedded GPU supports up to 16 texture coordinates to be calculated in a vertex shader. In this optimization step, we use 16 texture coordinates from the vertex shader and other 9 coordinates are calculated inside the fragment shader. As shown in Fig. 2, variable *gaussian55* and *g_value* have their numerical range between $[-2.0, 2.0)$ for the entire length of the program. As a result, these two variables can be assigned a *lowp* precision modifier to accelerate the computation process. All color read from texture memory can be assigned *lowp* precision. Using precision control lowers the instruction count significantly and achieve 2x speedup when compared to previous load sharing optimization.

```
uniform sampler2D  sTexture;
uniform mediump float width;
uniform mediump float height;
uniform mediump float filter_size;
varying mediump vec2  TexCoord;
const mediump mat3 gaussian55 = mat3(0.1502, 0.0952, 0.0256,
                                     0.0952, 0.0586, 0.0146,
                                      0.0256, 0.0146, 0.0037);
void main()
{
 mediump float offsetX = 1.0 / width;
 mediump float offsetY = 1.0 / height;
 mediump float i, j;
 mediump int a_i, a_j;
 mediump vec3 g_value = vec3 ( 0.0, 0.0, 0.0 );
 mediump vec2 Coord;
 for ( i = -2.0 ; i < 3.0; i++ )
      for ( j = -2.0 ; j < 3.0; j++ )
      {
         a_i = int ( abs(i) );
         a_j = int ( abs(j) );
         Coord = TexCoord + vec2(i * offsetX, j * offsetY);
         g_value += texture2D ( sTexture, Coord) ).rgb * gaussian55[a_i][a_j];
      }
 gl_FragColor = vec4 ( g_value, 1.0 );
}
```

**Fig. 2** 5×5 Gaussian blur fragment shader. Unoptimized version.

**Table 1** Fragment Shader Optimization. Execution time (ms) for 5×5 Gaussian filter. GPU is POWERVR SGX 540 at 200 MHz. Instruction count is calculated using PVRUniSCoEditor shader text editing tool [13].

| Optimization | Instruction Count | Execution Time (ms) |
|---|---|---|
| Basic | 288 | 537.63 |
| Loop Unroll | 181 | 105.93 |
| Load Sharing | 150 | 90.25 |
| Precision Control | 69 | 48.90 |

## 6. Design and Implementation of Algorithms

In this paper, we select three target algorithms (cartoon-style NPR, speeded-up feature detector (SURF), and stereo matching) to implement and analyze on the embedded GPU. These algorithms include multiple image processing routines such as Gaussian smoothing, bilateral filtering, color conversion, edge detection, etc. The implementation on the embedded GPU is optimized based on the techniques described in the previous section.

### 6.1 Cartoon-Style Non-photorealistic Rendering (NPR)

We present an implementation of the cartoon-style NPR algorithm using vertex and fragment shader units of a programmable embedded GPU. Given the input image $f(x)$, this image is convolved by a Bilateral filter kernel. A bilateral filter has a property of edge preserving smoothing [15], which is defined by

$$G[x] = \frac{\sum_{\hat{x}\in N} e^{-\frac{1}{2}\left(\frac{\|\hat{x}-x\|}{\sigma_d}\right)^2} w(x,\hat{x}) f(x)}{\sum_{\hat{x}\in N} e^{-\frac{1}{2}\left(\frac{\|\hat{x}-x\|}{\sigma_d}\right)^2} w(x,\hat{x})}, \tag{1}$$

where $x$ is a pixel location, $\hat{x}$ are neighboring pixels, $N$ is the kernel size, and $\sigma_d$ is the geometric spread (low-pass filtering). The range weighting function, $w(\cdot)$, behaves such that the weight is small for pixels in different regions with large contrasts. Therefore, there is much less smoothing across the edge between the regions. A popular choice for $w(\cdot)$ is given by

$$w(x,\hat{x}) = e^{-\frac{1}{2}\left(\frac{\|f(x)-f(\hat{x})\|}{\sigma_r}\right)^2}. \tag{2}$$

where $\sigma_r$ is the photometric spread in the image range. Next, the highlighting edges are overlaid to increase local contrast and sharpen the resulting cartoon-style image. Bilateral filtering is applied to the Y (Luminance) channel only, since it carries the majority of information about the image. Also, edge artifacts and noise are mostly seen in this channel.

#### 6.1.1 Implementation on the Embedded GPU

On an embedded GPU such as POWERVR SGX 540, exp2

(base-2 exponential) execute in a single instruction as compared to 4 instructions for exp (base-e exponential). As a result, we first modify Eq. (1) to utilize to exp2 instead of exp.

$$G^{'}[x] = \frac{\sum_{\hat{x}\in N} h(x,\hat{x})w(x,\hat{x})f(x)}{\sum_{\hat{x}\in N} h(x,\hat{x})w(x,\hat{x})}, \quad (3)$$

$$w(x,\hat{x}) = 2^{-c(\|f(x)-f(\hat{x})\|)^2}, c = -log_2 e/2\sigma_r^2, \quad (4)$$

where $h(x,\hat{x}) = e^{-\frac{1}{2}\left(\frac{\|\hat{x}-x\|}{\sigma_d}\right)^2}$, is a pre-computed Gaussian space function. The above technique is effective for all embedded GPU architectures supporting OpenGL ES 2.0.

The GPU acceleration is implemented in two stages. The first stage contains two rendering passes. In the first pass, the fragment shader program converts the input texture image from RGB to YCbCr color space. In the second pass, the pixel shader performs bilateral filtering by fetching 24 neighboring pixels, 16 of which are computed in the vertex shader program and passed to the fragment program using *varying* variables. Next, the second pass is iteratively rendered multiple times to produce the desired level of abstraction. After the first stage, the Sobel edge detection is employed to highlight edges in the abstracted luminance map. Finally, the YCbCr values are transformed back to the RGB color space and are rendered to a 32-bit RGBA texture or the screen buffer.

### 6.2 Stereo Matching

In this work, we adopt the belief propagation (BP) algorithm for depth estimation [16]. There are several BP proposals in literature. Majority of them focus on CPU implementations, where the execution time cannot satisfy low latency requirements. There are some BP algorithms that have been implemented on GPUs [17], but these implementations use high end PC GPU and are not targeted at handheld devices.

#### 6.2.1 Implementation on the Embedded GPU

The number of iteration and levels required to obtain a good quality disparity map are empirically set. The number of disparity is set at 16. The BP algorithm consists of the following blocks.

(1) RGB to Gray

In the first pass, the fragment shader program converts left and right images from RGB to gray color domain.

(2) Data cost calculation

Pixel coordinate $(i, j)$ in the left image is compared to $(i + d, j)$ in the right image. Data cost at depth $d$ is calculated as

$$val = \|L(i,j) - R(i+d,j)\|, \quad (5)$$

$$C(i,j,d) = (\lambda * min(val, DATA\_K)), \quad (6)$$

where, $L$ and $R$ are left and right image, respectively, $C$ is the data cost at depth $d$, and $\lambda$ and $DATA\_K$ are constant values. A fragment shader is limited by the maximum of 4 output values, each 1 byte in size. To perform the required operations for 16 disparity values, it requires 4 rendering passes, each rendering 4 values to a 32-bit RGBA texture.

(3) Message passing module

This module is the core of the algorithm. It reads the initial data cost and performs the message passing to obtain the final disparity map. In belief propagation, each pixel (i,j) outputs 4 different messages to the pixel surrounding it (up, down, left, and right). In order to calculate each of these messages, the pixel gets three messages corresponding to the surrounding pixels. For example, if we want to calculate the message in "up" direction, the pixel gets messages from (i,j+1), (i+1,j), (i-1,j) and sum them together with C(i,j). After this process, the final value is truncated and normalized. Algorithm 1 describes the pseudo code for message passing algorithm.

In the GPU implementation, the message passing module is divided into two stages. In the first stage, temporary messages for each of up, down, left, and right directions, are obtained by means of the three messages corresponding to the surrounding pixels and the center data cost. The temporary messages for each direction and 16 disparity values are rendered to 4 different 32-bit RGBA textures (pass 1 textures). In the second stage, the temporary messages for each direction are independently truncated and normalized. For each direction, the output values for 16 disparity values are rendered to 4 different 32-bit RGBA textures (pass 2 textures). These pass 2 textures serve as input to the first stage in the next iteration.

The above message passing algorithm is executed for a set number of iterations (0 to ITER-1) and levels (0 to LEVELS-1). At each level, starting from level 1, the rendered width and height are reduced by half.

**Algorithm 1** Message Pass

**First loop**, with d from 0 to 15. In the following pseudo code, we calculate the message for up direction
// Pass 1 Textures
msg(i,j,d) = msg(i,j+1,d) + msg(i+1,j,d) + msg(i-1,j,d) + data_cost(i,j,d)
if((msg(i,j,d) - msg(i,j,d-1)) > 1)
msg(i,j,d) = msg(i,j,d-1)
if((msg(i,j,d) - msg(i,j,d-1)) < -1)
msg(i,j,d-1)= msg(i,j,d)

**Second loop**, with d from 0 to 15 truncate the message below a value given by minimum value plus a constant DISC_K:
minimum += DISC_K
if(minimum < msg(i,j,d)
msg(i,j,d) = minimum
value += msg(i,j,d)

**Third loop**, with d from 0 to 15 normalizes the final message
value = value / 16
msg(i,j,d) = msg(i,j,d) - value // Pass 2 Textures

(4) Output

This module performs the summation of the final messages and the initial data cost of every pixel. Then, it calculates the minimum value among 16 disparity values. The minimum value is scaled to cover the range from 0 to 255.

### 6.3 Speeded-Up Robust Feature (SURF)

In this work, we have chosen the SURF [18] algorithm because of its favorable computational characteristics for parallel implementation and its state-of-the-art matching performance.

SURF algorithm locates features using an approximated method in obtaining the determinant of the Hessian. It replaces the second order Gaussian filters with a box filter approximation. Box filters can be evaluated extremely efficiently using the *integral image*. Given an integral image, the sum over any arbitrary sized 2D region can be computed in just four memory lookups. To achieve scale invariance, the filters are computed at a number of different scales $s$, and $3 \times 3 \times 3$ local maxima in scale and position determine the detected features. In our implementation, we do not compute orientation for algorithm simplicity. Once the position and the scale have been determined, a feature descriptor is computed, which is used to match features across images. Feature descriptor is built from a set of Haar responses computed in a $4 \times 4$ grid of sub-regions of a square of size $20s$ around each feature point. Twenty-five 2D Haar responses $(d_x, d_y)$ are computed using filters of size $2s \times 2s$ on a $5 \times 5$ grid inside each sub-region and weighted by a Gaussian with $\sigma = 3.3s$ centered at the interest point. Each sub-region constructs a four-dimensional vector $\mathbf{v} = (\sum d_x, \sum d_y, \sum |d_x|, \sum |d_y|)$ from these responses. Combining the vectors $\mathbf{v}$ from each sub-region yields a single 64-dimensional descriptor.

#### 6.3.1 Implementation on the Embedded GPU

(1) Integral Image Computation

The integral image is used to compute box filter and Haar filter responses at arbitrary scales. Since it must be computed over the entire image, it is quite expensive. Embedded GPU such as POWERVR SGX 540 supports only 32-bit RGBA texture (four 8-bit outputs). The 31 bits of precision available in RGBA texture have sufficient accuracy for images less than $2^{23}$ pixels in size (about $2048 \times 2048$). In the fragment shader program, we split the output sum value into a four component vector of 8-bit precision floats. Integral image is built using 2D reduction technique. The fragment shader adds the adjacent pixel values obtained from previous rendering pass as follows.

$$\begin{aligned} p_r(i,j) = {} & p_{r-1}(i,j) + p_{r-1}(i-2^{r-1}, j) \\ & + p_{r-1}(i, j-2^{r-1}) \\ & + p_{r-1}(i-2^{r-1}, j-2^{r-1}), \end{aligned} \tag{7}$$

where $p_r(i,j)$ is the sum value in the $r^{th}$ rendering iteration ($r \geq 1$) and $p_0$ represents the input gray scale image. Figure 3 shows the reduction scheme for a $8 \times 8$ image block. For a 800×480 resolution, this stage requires 10 rendering passes.

(2) Hessian Determinant

After constructing the integral image, we turn to the evaluation of box filters, which are used to locate interest points. The layout of the box filters is illustrated in Fig. 4 (a). The box filter size at different octaves (1 to 4) used in this implementation is given in Fig. 4 (b). First, we compute the box filter response for each filter size at the image resolution. Next, we use the hardware bilinear interpolation to generate filter responses at different octaves. This process requires 10 rendering passes for 10 different filter size values shown in Fig 4 (b). In a single rendering pass, the fragment shader program involves 32 texture lookup. The determinant and the laplacian values are rendered to RG components of a 32-bit RGBA component.

(3) Non-maximum Suppression (NMS)

Once the Hessian determinant values have been computed, the local maxima for a triplet of scales, over a given threshold value become interest points. Figure 4 (b) shows the triplet of scales used for calculating the local maxima. We perform $3 \times 3 \times 3$ NMS filtering in order to isolate the features. The process involves keeping a center value as interest point if this is the maximum value among 26 neighboring locations. As shown in Fig. 4 (b), a total of 8 rendering passes are required, one for each triplet of scales. The rendered width and height depends on the octave number in which a particular triplet lies.

(4) Point Table Generation

After NMS filtering, the coordinates of the interest points are extracted from the image and assembled into a table. Creating the interest point table using a fragment shader suffers from extra computations in calculating the indices. In

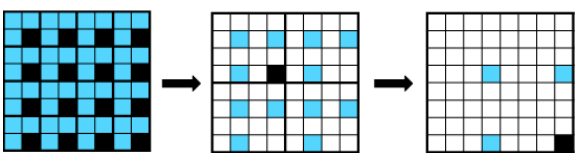
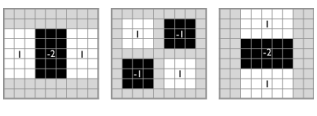


**Fig. 3** An example of 2D reduction algorithm in a 8×8 region. The sum of gray value at blue and black pixel locations is rendered to the black pixel location.




(a)

| Octave | Filter Size | | | |
|---|---|---|---|---|
| First | 9 | 15 | 21 | 27 |
| Second | 15 | 27 | 39 | 51 |
| Third | 27 | 51 | 75 | 99 |
| Fourth | 51 | 99 | 147 | 195 |

(b)

**Fig. 4** Box filters. (a) Approximation for the second order Gaussian partial derivatives using box filters. (b) Box filter size at different octaves.

this work, feature points are extracted from the NMS filtering output and composed in a point table. The NMS filtering output is downloaded to CPU buffer from the rendered texture using `glReadPixels()`. All threads running on the GPU are thus synchronized. Feature point locations are extracted from the downloaded buffer and are put together in a point table as a two-dimensional texture of size 340×30, as shown in Fig. 5. Note that we limit the maximum number of interest points to 300 for a 800×480 resolution image.

(5) Haar Responses

The feature descriptor construction requires computing hundreds of Haar filter responses. This stage is implemented in 5 rendering passes. In the first pass, we extract 20×20 region around each feature point. This region is divided into a 4×4 grid of sub-regions. For each sub-region twenty-five 2D Haar responses $(d_x, d_y)$ are calculated. In this pass, the fragment shader output $(d_x, d_y, sign(d_x), sign(d_y))$ is rendered to a $200 \times 600$, 32-bit RGBA texture.

In the second pass, the Haar response corresponding to each 5×5 region is accumulated to generate $(\sum d_x, \sum |d_x|, \sum d_y, \sum |d_y|)$. Accumulated output from each 5×5 region is rendered to two adjacent pixels in the output texture. In the third pass, the values stored in two adjacent pixels are squared, grouped and rendered to a single pixel value (RGBA component). Next, the values obtained from the previous pass are summed over a 4×4 region and rendered to a 10×30, 32-bit RGBA texture. In the final rendering pass, the output values obtained in the second rendering pass are normalized.

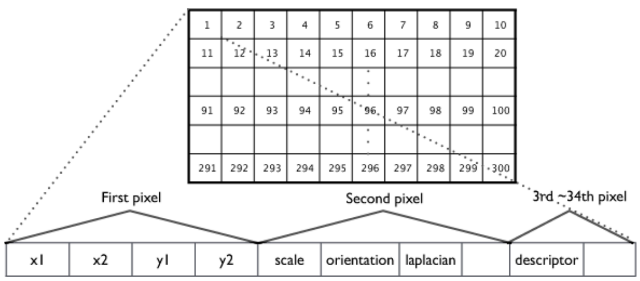


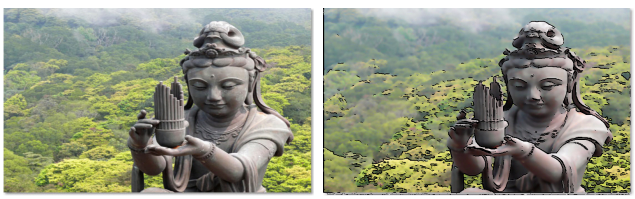

**Fig. 5** Interest point table.

## 7. Experimental Results

In this section, we compare the CPU and GPU implementations of algorithms described in Sect. 6. Performance is evaluated in terms of the execution time and the speedup achieved. The algorithms are implemented on a state-of-the art smartphone with ARM CORTEX A8 CPU (1 GHz) and POWERVR SGX 540 GPU (200 MHz). Table 2 shows the acceleration results for each of the algorithms. In the following subsections, we evaluate each of three algorithms separately.

### 7.1 Cartoon-Style NPR

Figure 6 shows an example of cartoon-style NPR. The number of bilateral filtering iterations used in our experiment is two. As depicted in Fig. 6, the size of the bilateral filter decides the level of abstraction in the rendered output. In terms of the execution time, the GPU implementation achieves a speedup of 5~6x when compared to the CPU counterpart. Note that the CPU implementation is a fixed point (integer) implementation using lookup tables for the **exp** function. The bilateral filter fragment shader program involves large number of dependent texture lookups, which force the GPU threads architecture to stall the parallel operations and execute the instructions sequentially. This results in bottleneck and limits the speedup achieved.



(a) (b)

**Fig. 6** An example of cartoon-style NPR. (a) The original image at the resolution 800×480. (b) 5×5 Bilateral filter result.

**Table 2** Execution time comparison in millisecond. CPU is ARM CORTEX A8 running at 1 GHz. GPU is POWERVR SGX 540 at 200 MHz.

| Algorithm | Parameters | CPU | GPU | | | Speed-up |
|---|---|---|---|---|---|---|
| | | | CPU to GPU | GPU to CPU | Kernel Execution | |
| Cartoon-style NPR | 800×480, size = 5×5 | 1545.7 | 7 | 46.34 | 242.7 | 5.22x |
| Stereo matching | 384×288, iter = 2 | 4152 | 4 | 12.57 | 578 | 6.98x |
| | 384×288, iter = 4 | 6976 | 4 | 12.57 | 1086 | 6.32x |
| | 384×288, iter = 8 | 12161 | 4 | 12.57 | 2083 | 5.79x |
| | 384×288, iter = 12 | 17413 | 4 | 12.57 | 3125 | 5.54x |
| | 384×288, iter = 20 | 27889 | 4 | 12.57 | 5263 | 5.28x |
| SURF | 800×480 | 1703 | 7 | 63.22 | 942.5 | 1.68x |

(a) (b) (c) (d)

**Fig. 8** Examples of SURF feature detection. (a) Original image (800×480). (b) Detected SURF features. (c) Original image (800×480). (d) Detected SURF features.

(a) (b) (c)

**Fig. 7** An example of BP stereo matching. (a) Tsukuba image (left) at 388×244. (b) Ground truth. (c) GPU implementation result with 4 levels and 15 iterations.

### 7.2 Stereo Matching

Figure 7 shows the BP stereo matching results with 4 levels and 15 iterations. In Table 2, the CPU fixed point implementation is compared to the GPU implementation. The CPU implementation is done using C programming language and interfaced (JNI) with the Java layer using Android NDK. The GPU implementation achieves a speedup factor in the range of 5∼7x. Both the CPU and GPU implementation heavily suffers from memory bandwidth issue. High memory access intensity coupled with intensive logical operations significantly increase the execution time. Additionally, the GPU implementation is bottlenecked by large number of rendering passes, which reduces the maximum speedup achieved. The main reason for large number of rendering passes is the limited number of output values (4), which can be rendered in a fragment shader program. For example, calculating data cost for 16 disparity levels costs four rendering cycles, because of this limitation.

### 7.3 SURF

Figure 8 shows the GPU implementation results for input image of resolution 800×480. The GPU implementation in case of SURF achieves the smallest speedup, when compared to the CPU counterpart. Similar to stereo matching, the CPU implementation is developed using C programming language and Android NDK. The lack of support for higher bit-precision textures (floating point) in SGX 540 is one of the major reason for low speedup when implemented on the GPU. The overhead incurred in grouping and splitting the integral image values into RGBA texture components is one such bottlenecks. Table 3 shows the breakup of GPU time for different modules. As shown, the Hessian determinant

**Table 3** Execution time breakdown in millisecond for SURF algorithm.

| Step | On CPU | On GPU |
|---|---|---|
| (Step 1) Texture Uploading | - | 7 |
| (Step 2) RGB to Gray, Integral Image | 525 | 292 |
| (Step 3) Hessian Determinant | 759 | 557 |
| (Step 4) Non-maximum Suppression | 159 | |
| (Step 5) Point Table Creation | – | |
| (Step 6) Feature Descriptor Extraction | 260 | 93.5 |
| (Step 6) Texture Downloading | - | 63.22 |
| **Total** | 1703 | 1012.72 |

is the most complex part when implemented on an embedded GPU. This is followed by integral image computation and feature descriptor extraction modules, respectively. The Hessian determinant calculation takes the maximum execution time as it suffer intensively from 32 dependent texture lookups in a single rendering pass. In addition, designing GPU implementation for multiple octaves and scales, results in large number of rendering cycles. This inherently limits the maximum speedup achieved.

## 8. Conclusion

In this paper, we explored the implementation, optimization, and evaluation of image processing and computer vision algorithms on the embedded GPU using OpenGL ES 2.0 shading language. In addition to characterizing the embedded GPU, we proposed optimization techniques for efficient shader design. We selected three algorithms namely, cartoon-style NPR, stereo matching, and SURF. Based on the proposed optimization techniques, these algorithms were implemented on an embedded GPU.

## Acknowledgement

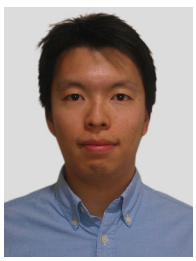
This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2011-0003392). This work was supported by Samsung Electronics.

## References

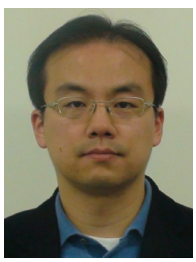

[1] General Purpose GPU Programming (GPGPU) Website. http://www.gpgpu.org
[2] N. Singhal, I.K. Park, and S. Cho, "Implementation and optimization of image processing algorithms on handheld GPU," Proc. IEEE International Conference on Image Processing, pp.4481–4484, Sept. 2010.
[3] R.J. Rost, OpenGL Shading Language, Second ed., Addison-Wesley Professional, 2006.
[4] NVIDIA Corporation, Compute Unified Device Architecture (CUDA). http://developer.nvidia.com/object/cuda.html
[5] Khronos Group, Open Computing Language. http://www.khronos.org/opencl/
[6] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips, "GPU computing," Proc. IEEE, vol.96, no.5, pp.879–899, 2008.
[7] N.M. Cheung, O.C. Au, M.C. Kung, and P.H.W. Wong, "Highly parallel rate-distortion optimized intra-mode decision on multicore graphics processors," IEEE Trans. Circuits Syst. Video Technol., vol.19, no.11, pp.1692–1703, Nov. 2009.
[8] Y. Allusse, P. Horain, A. Agarwal, and C. Saipriyadarshan, "GpuCV: An opensource GPU-accelerated framework for image processing and computer vision," Proc. ACM International Conference on Multimedia, pp.1089–1092, Oct. 2008.
[9] P. Babenko and M. Shah, "MinGPU: A minimum GPU library for computer vision," Real-Time Image Processing, vol.3, no.4, pp.255–268, Dec. 2008.
[10] J. Fung, S. Mann, and C. Aimone, "OpenVIDIA: Parallel GPU computer vision," Proc. ACM International Conference on Multimedia, pp.849–852, Nov. 2005.
[11] NVIDIA NPP Library. http://www.nvidia.com/object/npp.html
[12] I.K. Park, N. Singhal, M.H. Lee, S. Cho, and C.W. Kim, "Design and performance evaluation of image processing algorithms on GPUs," IEEE Trans. Parallel Distrib. Syst., vol.22, no.1, pp.91–104, Jan. 2011.
[13] IMAGINATION Technologies, POWERVR SDK. http://www.imgtec.com
[14] T.A. Moller and J. Strom, "Graphics processing units for handhelds," Proc. IEEE, vol.96, no.5, pp.779–789, May 2008.
[15] S. Paris, P. Kornprobst, J. Tumblin, and F. Durand, "Bilateral filtering: Theory and applications," Foundation and Trends in Computer Graphics and Vision, vol.4, no.1, pp.1–73, 2009.
[16] Q. Yang, L. Wang, R. Yang, H. Stewénius, and D. Nister, "Stereo matching with color-weighted correlation, hierarchical belief propagation and occlusion handling," IEEE Trans. Pattern Anal. Mach. Intell., vol.31, no.3, pp.492–504, March 2009.
[17] Q. Yang, L. Wang, R. Yang, S. Wang, M. Liao, and D. Nistér, "Real-time global stereo matching using hierarchical belief propagation," Proc. British Machine Vision Conference, pp.989–998, Sept. 2006.
[18] H. Bay, A. Ess, T. Tuytelaars, and L.V. Gool, "SURF: Speeded up robust features," Comput. Vis. Image Understand., vol.110, no.3, pp.346–359, June 2008.


**Nitin Singhal** received the M.S. degree in electrical engineering and computer science from Seoul National University, Seoul, Korea in 2008 and B.S. degree in Electronics and Communication Engineering from Indian Institute of Technology (IIT), Guwahati, India in 2006. He is presently working at Samsung Electronics Co. Ltd., Suwon, Korea and is affiliated with the Digital Media and Communication R&D center. He is an active member of IEEE. His research interests include computer vision, computational photography, GPU computing, and digital right management.

**Jin Woo Yoo** received the B.S. degree in information and communication engineering from Inha University, Incheon, Korea, in 2010. Currently he is working toward M.S. degree in information and communication engineering in Inha University. His research interests include computational photography, feature extraction, and GPGPU for image processing and computer vision.



**Ho Yeol Choi** received the B.S. degree in information and communication engineering from Inha University, Incheon, Korea, in 2010. Currently he is working toward M.S. degree in robot engineering in Inha University. His research interests include multi-view stereo reconstruction, motion deblurring, and GPGPU for image processing and computer vision.



**In Kyu Park** received the B.S., M.S., and Ph.D. degrees from Seoul National University in 1995, 1997, and 2001, respectively, all in electrical engineering and computer science. From September 2001 to March 2004, he was a Member of Technical Staff at Samsung Advanced Institute of Technology. Since March 2004, he has been with the School of Information and Communication Engineering, Inha University, where he is an associate professor. From January 2007 to February 2008, he was an exchange scholar at Mitsubishi Electric Research Laboratories (MERL). Dr. Park's research interests include the joint area of computer graphics and vision, including 3D shape reconstruction from multiple views, image-based rendering, computational photography, and GPGPU for image processing and computer vision. He is a member of IEEE and ACM.