편집위원 : 김홍국 (광주과학기술원)

# 스마트폰에서의 영상처리를 위한 GPU 활용

박인규, 최호열
인하대학교

## 요 약

본 기고에서는 최근 스마트폰에서 요구되는 다양한 멀티미디어 어플리케이션을 embedded GPU(Graphics Processing Unit)를 이용하여 고속 병렬처리하기 위한 GPGPU (General-Purpose Computing on GPU) 기술 및 영상처리 분야의 응용 사례를 소개한다. 일반적인 데스크탑 컴퓨팅 환경과 달리 제약사항이 많은 embedded 환경에서의 GPGPU 응용 기술은 아직 초기단계이다. 그러나 급격히 발전하는 embedded GPU IP와 OpenCL과 같은 API의 등장으로 embedded GPU를 이용한 고속 병렬처리 환경이 수 년 이내에 일반화 될 것이다. 본 기고에서는 그 가능성을 점검하기 위하여 embedded GPU에서의 영상처리를 위한 최신 하드웨어와 소프트웨어 환경의 발전 동향을 소개한다. 더불어 최신 스마트폰에서의 GPGPU기술을 사용한 영상처리 사례와 영상처리 알고리즘의 GPGPU 알고리즘 구현시 고려해야 할 주요 사항을 정리한다.

## Ⅰ. 서 론

최근 스마트폰은 단순한 통신수단에서 영상, 음성, 통신이 결합된 개인용 멀티미디어 기기로 발전하였다. 특히 고화질의 디스플레이, 카메라, 중앙처리장치 성능의 발달로 인해 사용자들은 더욱 선명하고 실감나는 영상을 즐기고 있다. 하지만 이러한 변화는 스마트폰의 중앙처리장치가 처리해야 할 데이터와 연산의 양이 역시 비약적으로 증가한다는 것을 의미한다.

모바일 기기의 특성상 스마트폰은 데스크탑 환경과는 근본적으로 다른 제약 조건을 가진다. 우선 배터리를 사용하므로 한정된 전력 내에서 필요한 컴퓨팅 성능을 확보해야 하는 어려움이 있다. 최근 스마트폰의 응용프로세서 (Application Processor, AP)의 발전방향은 전력소모를 줄이기 위하여 멀티코어를 지향하고 있다. 삼성전자의 Exynos 5250과 같은 최신 응용프로세서는 2.0GHz로 동작하는 Cortex A15 듀얼 코어를 갖추고 있으며 이는 수년전의 노트북 CPU 수준의 성능과 유사하다. 그러나 이러한 성능은 스마트폰의 수백만 화소의 영상취득장비로부터 얻어지는 대용량의 데이터를 실시간으로 처리하기 위해서는 한계가 존재한다. 따라서 스마트폰에서 멀티미디어 데이터를 처리하기 위한 새로운 해결책이 필요로하게 되었다. 스마트폰에 탑재되는 최신 응용프로세서는 CPU IP외에도 GPU IP를 포함하는 경우가 일반적이므로, embedded GPU를 이용하여 CPU의 계산 부담을 분산하면서 GPU의 고속 처리기술을 응용한다면 매우 적절한 대안이 될 수 있다. 또한 영상처리와 같은 분야는 알고리즘의 속성이 화소당 독립적인 연산을 수행하는 경우가 많으므로 GPU 병렬처리에 매우 적합하다. 또한 최신 embedded GPU IP 의 코어의 개수는 지속적으로 증가하고 있으며 12개의 코어를 갖는 고성능 제품도 출시되고 있다.

GPGPU기술은 embedded GPU의 연산능력을 3차원 그래픽스 연산이 아닌 일반적인 용도로 사용하는 것이다. 모바일 환경에서 GPGPU기술을 사용함에 따라 응용프로세서의 부담을 줄여줌과 동시에 병렬처리를 통해 연산속도를 가속화 하는 효과를 얻을 수 있다. PC 플랫폼의 GPU의 경우 CUDA, OpenCL, OpenGL Shading Language (GLSL), Cg 등의 API를 통하여 GPGPU 기술을 구현할 수 있다. 그러나 현재 일반적인 embedded 환경에서의 GPGPU 를 가능하게 하는 API는 OpenGL ES 2.0 Shading Language 뿐이다. 한편 최신 embedded GPU IP는 보다 범용적인 병렬처리 API인 OpenCL을 지원한다. 삼성전자나 TI, 애플 등에서 생산하는 최신 응용프로세서는 OpenCL full profile을 지원할 예정이며, 다양한 멀티미디어 응용 소프트웨어가 embedded GPU를 기반으로 고속 동작할 것이다.

본 기고의 구성은 다음과 같다. 제II장에서는 최신 스마트폰에서 사용되는 embedded GPU IP의 기술발전 동향을 살펴보고, 제III장에서는 embedded GPU에서의 GPGPU를 위한 병렬처리 API를 소개한다. 제 IV장에서는 실제 embedded GPU를 이용한 병렬 영상처리 사례를 소개하며 여러 가지 최적화 기법을 제시한다. 제 V장에서는 본 기고의 결론을 맺는다.

# Ⅱ. Embedded GPU

모바일 프로세서의 발전방향은 저전력 고성능 컴퓨팅이다. 이러한 목표를 달성하기 위해서 모바일 프로세서 기술은 멀티코어 형태로 발전해 가고 있다. 예를 들어 NVIDIA의 최신 응용프로세서인 Tegra3의 경우 쿼드코어 ARM Cortex A9 CPU와 최대 12개의 코어를 가진 ULP GeForce GPU를 갖추고 있다.

## 1. 주요 Embedded GPU IP

모바일 프로세서는 SoC (System On Chip) 형태로 응용프로세서에 CPU와 GPU가 함께 내장되어 있다. 내장된 CPU는 스마트폰의 주 프로세서로 동작하게 된다. 대표적인 응용프로세서로서 TI(Texas Instrument)의 OMAP, 삼성전자의 Exynos, 퀄컴의 Snapdragon, Apple의 A5가 있다. 이들은 CPU IP는 모두 ARM사의 Cortex 시리즈를 사용하고 있다. 각 응용프로세서 관련 특징 및 기술 동향은 [1]에 상세하게 정리되어 있다.

최신 AP인 A5X와 Exynos 5250의 경우 각각 2,048×1,536 및 2,560×1,600크기의 해상도를 지원하므로 기존의 싱글/듀얼코어GPU로는 렌더링 속도 확보에 어려움을 갖는다. 따라서 이와 같은 최신 AP는 쿼드코어를 장착한 embedded GPU를 채택하는 추세이다.

Embedded CPU IP와는 대조적으로 embedded GPU IP는 다양한 제조사의 제품이 채택되고 있으며, Imagination Technologies의 PowerVR SGX, ARM의 Mali, Qualcomm의 Adreno, NVIDIA의 ULP (Ultra Low Power) GeForce 시리즈 등이 대표적이다. 2011년 상반기까지의 각 GPU IP의 하드웨어적 특성은 [2]를 참고하기 바란다. 삼성전자의 Exynos 5250의 경우 쿼드코어 GPU, OpenCL 지원, 12.8 GB/s의 높은 메모리 전송 대역폭의 특징을 가지고 있어 GPGPU에 적합한 구조를 가지고 있다. 〈표 1〉에 제조사별 주요 응용프로세서의 CPU IP와 GPU IP를 정리하였다.

표 1. 주요 응용프로세서 에 탑재된 Embedded CPU와 GPU

| 응용프로세서 | 제조공정 | CPU | GPU |
|---|---|---|---|
| APPLE A5 | 45nm | 2×Cortex-A9@1GHz | PowerVR SGX 543MP2 |
| APPLE A5X | 45nm | 2×Cortex-A9@1GHz | PowerVR SGX 543MP4 |
| NVIDIA Tegra 2 | 40nm | 2×Cortex-A9@1GHz | ULP GeForce |
| NVIDIA Tegra3 | 40nm | 4×Cortex-A9@1.5GHz 1×Cortex-A9@500Mhz | ULP GeForce |
| Samsung Exynos 4210 | 45nm | 2×Cortex-A9@1.2GHz | ARM Mali-400 MP4 |
| Samsung Exynos 5250 | 32nm | 2×Cortex-A15@2GHz | ARM Mali-T604 MP4 |
| TI OMAP 4430 | 45nm | 2×Cortex-A9@1.2GHz | PowerVR SGX 540 |
| Qualcomm MSM8x60 | 45nm | 2×Scorpion 1.2GHz | Adreno 220 |

## 2. GPGPU관점에서의 Embedded GPU의 특징

GPGPU관점에서 embedded GPU를 사용할 때 CPU와 GPU 간의 데이터 전송이 빈번하기 때문에 낮은 전송 대역폭은 GPU 병렬처리의 효율을 떨어뜨리는 주원인이다. 〈그림 1〉은 삼성전자 S5PC110 응용프로세서에서 ARM Cortex A8 CPU(1GHz)와 PowerVR SGX 540 GPU(200MHz)간의 메모리 전송 대역폭의 측정 결과이다. CPU에서 GPU로 데이터를 전송할 경우 약 220.54MB/sec의 대역폭을 가지며 GPU에서 CPU로의 데이터 전송의 경우 30.92MB/sec의 대역폭을 가진다. 데스크탑 컴퓨터의 PCI Express 버스가 16GB/sec의 대역을 가지는 것을 감안하면 매우 협소한 대역이라고 할 수 있다. 따라서 embedded GPU에서의 알고리즘 개발은 CPU/GPU간 메모리 전송량의 최소화가 가장 중요한 문제이다.

Embedded GPU의 또다른 특징은 부동소수점 연산에 특화된 하드웨어라는 점이다. Embedded CPU는 하드웨어적으로 부동

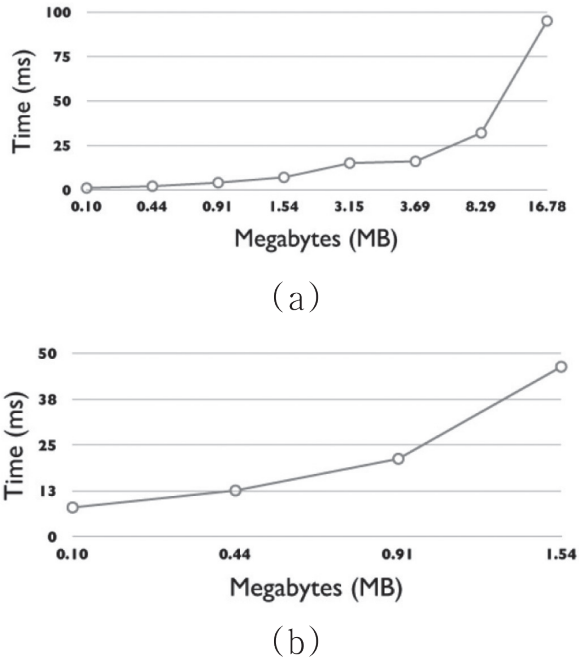



그림 1. Embedded CPU와 GPU간의 데이터 전송률 (ARM Cortex A9 CPU와 PowerVR SGX 540을 장착한 삼성전자 S5PC110에서의 사례) [1] (a) CPU 에서 GPU, (b) GPU에서 CPU

소수점 연산을 지원하지 않는 경우가 많고, VFPv3 구조와 같이 최신 CPU IP가 지원하는 부동소수점 연산이라 하더라도 여전히 연산을 위한 instruction cycle 수가 많다. 따라서 고정소수점 연산과 비교하였을 경우 이득을 얻기 어렵고 embedded GPU의 부동소수점 연산능력에 비해 현저하게 떨어지는 성능을 보인다.

이를 확인하기 위하여 5×5 Gaussian 필터링을 반복 수행하는 간단한 실험을 수행하였다. 이 경우 CPU(ARM Cortex A8)에서 부동소수점 구현은 3,075ms의 수행 속도를 보인다. 그러나 모든 연산을 고정소수점 연산으로 변환할 경우 207ms로 속도가 현저히 증가하게 된다. 반면에 GPU(PowerVR SGX 540 GPU)에서의 병렬구현의 경우 수행시간은 49ms로서 CPU에서의 고정소수점 연산에 비해서도 월등한 성능을 보인다.

## Ⅲ. Embedded GPU를 위한 GPGPU 소프트웨어 환경

Embedded GPU를 이용한 GPGPU 기술을 사용하기 위해서는 GPU를 프로그래밍 할 수 있는 API가 필요하다. 현재 모바일 환경에서 대표적인 프로그래밍 API는 OpenGL ES 2.0 Shading Language이다. 최근에 범용적인 목적으로 embedded GPU를 포함한 다양한 컴퓨팅 유닛에 접근하는 통합적인 API인 OpenCL이 새로운 표준으로 개발되었다. 그러나 현재 출시되어 있는 embedded GPU는 응용프로세서 수준에서 아직 OpenCL을 지원하고 있지 않는 상황이다. 그러나 OpenCL은 머지않은 미래에 embedded 환경에서 GPGPU를 위한 병렬처리 API로 사용될 것이 확실시 되고 있으므로 본 장에서는 OpenGL ES Shading Language와 OpenCL에 대해 모두 알아본다.

### 1. OpenGL ES Shading Language [4]

OpenGL ES 2.0은 OpenGL 2.0의 embedded 버전으로서 embedded GPU에서 shader를 사용한 programmable hardware 구성을 위한 API이다. OpenGL ES 1.0, 1.1의 고정 파이프라인에 비해 OpenGL ES 2.0은 vertex shader와 fragment shader를 사용한 프로그래밍 가능한 파이프라인을 지원하며, 각 shader는 Shading Language인 GLSL ES를 이용하여 프로그래밍 한다. 〈그림 2〉에 OpenGL ES 2.0의 파이프라인 구조를 도시하였다.

Shader를 위한 입력 데이터는 텍스처의 형태로 제공된다. Shader 프로그래밍을 통해 사용자가 작성한 파이프라인을 거쳐 계산된 결과는 프레임 버퍼로 출력된다. 그러나 GPGPU의 데이터 출력의 경우 off-screen rendering을 통하여 결과 데이터를 디스플레이에 출력하지 않고 데이터를 담고 있는 텍스처의 형태로 저장하는 것이 일반적이다. 이러한 과정을 통하여 embedded GPU를 그래픽스 렌더링 뿐만 아니라 일반적인 목적으로 사용이 가능하다 [5].

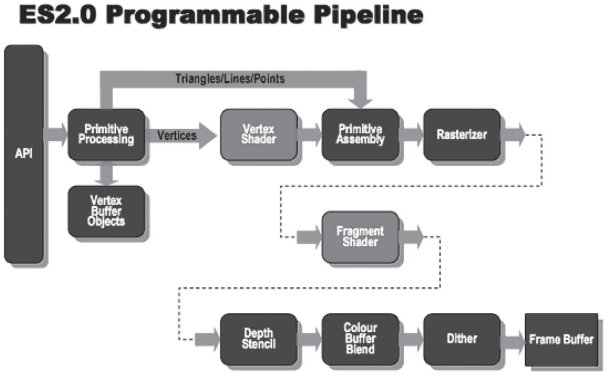



그림 2. OpenGL ES 2.0 Programmable 그래픽 파이프라인 [4]

Shading Language는 근본적으로 영상처리를 위한 GPGPU 용도로 만들어진 API가 아니므로 몇 가지 심각한 제약사항이 존재한다. [6]. 가장 큰 제약은 텍스처를 통한 데이터 입력과 출력이다. 반드시 텍스처를 이용해 데이터의 입출력이 이루어 지므로 개별적인 병렬 작업에서 생성된 결과는 반드시 한 화소의 컬러 값으로만 출력하여 저장할 수 있다. 이러한 점은 중복 계산 등의 비효율적인 알고리즘의 흐름을 야기시킬 뿐만 아니라 전역 동기화가 필요할 때마다 다른 shader 프로그램으로 작성해야 하므로 다수의 렌더링이 필요하게 될 것이다. 이 때 embedded GPU에서의 알고리즘 수행 속도는 렌더링 회수에 비례하여 감소하게 된다. 두 번째 제약사항은 shader 내부에서는 병렬 작업 간에 동기화를 수행할 수 없다는 점이다. 따라서 동기화가 필요한 경우 shader를 두 개의 별도의 shader 프로그램으로 나누어 렌더링을 두 번 수행해야 한다. 이는 역시 수행 속도를 저하시킨다.

### 2. OpenCL [7]

OpenCL은 개방형 범용 병렬 컴퓨팅 프레임워크로 다양한 플랫폼에서 사용 가능한 통합 API이다. OpenCL은 한 개의 host를 가지고 있으며 이는 여러 processing element를 제어한다. OpenCL은 data 병렬성, task 병렬성 등 다양한 형태의 병렬처리 방식을 지원한다. 이러한 병렬처리는 host가 kernel을 제어하는 방식으로 동작한다.

OpenCL을 이용하여 GPGPU 기반 프로그래밍을 수행하는

경우, 위에서 기술한 OpenGL ES의 제약사항이 극복이 되고, 그래픽 파이프라인에 구속되지 않고 자유롭게 병렬처리 프로그래밍이 가능하다는 장점이 존재한다. 특히 공유 메모리 사용, 병렬 thread간의 동기화, 가용 메모리 공간 전체에 대한 scatter 연산이 가능하므로 고속 구현 가능한 알고리즘의 범위가 크게 늘어날 것이다.

OpenCL에서 처리해야 할 데이터는 work-item, work-group에 의해서 나뉘어 데이터 병렬성 (data-parallelism)에 의해 처리된다. 일련의 병렬처리 과정들은 사용자가 작성한 context를 command queue에서 순차적으로 처리하는 방식으로 수행되게 된다. 이러한 구조는 SIMD, SPMD등의 다양한 방식의 병렬처리를 가능하게 할 뿐만 아니라 CPU, GPU, DSP등 서로 다른 processing element들 간의 병렬처리를 가능하게 한다.

## Ⅳ. 스마트폰에서의 GPU활용 영상처리 사례

Embedded GPU에서의 영상처리는 programmable 파이프라인구조에 적합한 알고리즘 설계뿐 아니라 사용하는 embedded GPU의 하드웨어 특징을 잘 파악하고 한정된 자원을 이용하여 최대의 효과를 이끌어 내는 것이 중요하다. 본 기고에서는 Imagination Technologies사의 PowerVR SGX IP를 플랫폼으로 한 알고리즘 구현 최적화 사례, 영상처리 알고리즘의 설계 및 구현 사례를 제시한다. 본 실험에서 사용된 스마트폰은 삼성전자의 갤럭시S 이다. 갤럭시S의 응용프로세서는 삼성전자의 S5PC110이며 ARM Cortex A8 1GHz와 PowerVR SGX 540을 내장하고 있다.

### 1. Shader 코드의 최적화 기법

〈그림 3〉은 Gaussian 필터링 알고리즘을 fragment shader로 구현한 기본 코드이다. 이러한 기본 코드는 loop unrolling, shader간 작업 분담, 부동소수점 정밀도 제어 등을 통해 최적화 할 수 있다.

Loop unrolling은 데스크톱 GPGPU 프로그래밍에서도 흔히 사용되는 최적화 기법으로서, for문이나 while문과 같은 반복문을 직접적인 명령어의 나열로 대치하는 기법이다. 즉 〈그림 3〉의 이중 for문은 5×5 Gaussian 커널을 통해 마스킹을 수행하는 부분인데 25회 반복되어 수행된다. 이에 대해 loop unrolling을 수행하면 for 문의 제어를 위한 증가 연산, 비교 연산 등이 생략되므로 계산량이 감소하게 된다.

```
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 flaot i, j;
    mediump flaot 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);
}
```

그림 3. 5×5 Gaussian 필터링의 Fragment Shader기본 코드 [3]

Shader는 vertex shader와 fragment shader로 나뉘어지며 영상처리의 경우 주로 화소당 연산을 수행하는 fragment shader를 통해 알고리즘을 구현하게 된다. 따라서 vertex shader의 작업 분담은 매우 작으며 화면에 꽉 차는 사각형 하나를 3차원에서 2차원으로 투영하는 정도의 간단한 일을 수행한다. 이 때 vertex shader에서 fragment shader로 16개의 변수를 전해줄 수 있다. 이를 이용하여 〈그림 3〉과 같은 경우 25개의 텍스처 좌표를 (변수 Coord) fragment shader에서 직접 계산하지 않고 그 중 16개는 vertex shader에서 계산하여 넘겨받으면 두 shader간의 작업 불균형이 개선되어 전체적인 성능이 향상된다.

OpenGL ES 2.0에서는 정밀도에 따라 highp, midiump, lowp와 같이 세 가지 형태의 부동소수점 숫자를 지원한다. 이는 각각 32비트, 16비트, 10비트 표현을 지닌다. 특히 lowp는 [-2.0, 2.0)의 범위에서 1/256의 정밀도를 지니며 연산에 있어서 highp에 비해 매우 적은 계산량을 가진다. 따라서 범위와 정밀도가 허용한다면 lowp를 이용한 부동소수점 표현의 활용이 바람직하다. 〈그림 3〉의 경우 g_value 및 gaussion55 배열의 원소는 연산의 범위가 [-2.0, 2.0)이고 1/256의 정밀도로 필터링 결과가 크게 열화되지 않으므로 lowp로 변경하여 연산량을 감소시킬 수 있다.

이와 같이 〈그림 3〉의 기본 코드를 3단계에 걸쳐 최적화한 개선 결과를 〈표 2〉에 제시하였다. 전체적으로 약 11배의 속도향상을 얻을 수 있다. 이 실험결과는 embedded GPU와 같이 제

표 2. 최적화에 따른 Shader 프로그램의 명령어 수 감소와 속도 향상 결과 (단위: ms) [3]

| 최적화 | 명령어 수 | 수행시간 |
|---|---|---|
| 초기 코드 | 288 | 537.63 |
| Loop Unrolling | 181 | 105.93 |
| Shader간 작업분담 | 150 | 90.25 |
| 정밀도 제어 | 69 | 48.90 |

한된 환경에서의 최적화의 중요성을 잘 보여주고 있다.

## 2. 사례1: Cartoon 스타일 NPR

Cartoon 스타일의 NPR (Non-Photorealistic Rendering)은 영상을 단순화, 강조 등의 효과를 통해서 회화적인 표현에 사용되는 영상처리 알고리즘이다. 〈그림 4〉에 그 사례를 제시하였다.

(a) (b)

그림 4. Cartoon Style NPR 결과 [3]
(a) 원본영상, (b) Cartoon Style NPR 적용결과

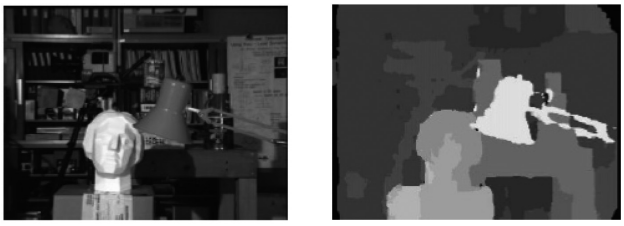
알고리즘은 세 단계로 이루어 진다. 첫 번째 단계는 RGB 영상을 YCbCr영상으로 변환하는 단계이다. 이 때 영상데이터는 텍스처의 형태로 shader의 입력으로 보내지며 fragment shader에서 텍스처의 화소당 컬러 공간 변환을 수행한다. 두 번째 단계에서는 bilateral 필터링을 반복적으로 수행하여 에지를 보존하면서 영상을 평활화(smoothing) 시킨다. 필터링의 반복 수행은 FBO (frame buffer object)를 이용하여 off-screen 렌더링의 형태로 구현할 수 있다. 이 과정에서 exponential 함수 등 다수의 부동소수점 연산이 필요하게 된다.

이 때 공간 함수는 lookup 테이블의 형태로 미리 계산을 해 둔다. 또한 전 절에서 설명한 shader간 작업 분담 개선을 적용하여 bilateral 필터링 수행시 16개의 화소 좌표는 vertex shader에서 구한 후 varying 변수를 통해 fragment shader로 넘겨주었다. 세 번째 단계는 입력 영상에 대해 Sobel 에지 검출을 수행하고 그 결과를 2단계의 결과 영상에 오버레이한다. 주어진 CPU와 GPU에서의 구현 결과, CPU에 비해 약 5배의 속도 향상이 이루어졌다.

## 3. 사례2: 양안 스테레오 정합

스테레오 정합은 두 장의 좌우 영상간의 정합을 통하여 disparity 지도를 추정하고 이에 의해 영상내의 화소의 상대적 깊이를 추정해 내는 기법이다. 단순한 윈도우 정합과 같은 지역적 알고리즘 보다 Graph Cuts 또는 Belief Propagation과 같은 전역적 기법에서는 영상의 화소간의 smoothness 제한조건에 의해 보다 정확한 disparity 지도 추정이 가능하다.

본 사례에서는 Belief Propagation (BP) 알고리즘[8]을 embedded GPU에서 구현하여 그 결과를 분석하였다. Belief Propagation 알고리즘은 각 단계에서의 연산이 데이터 의존성이 적은 특징을 가지고 있어 화소 단위의 병렬처리에 적합한 구조이다. 알고리즘은 3단계로 구성되며 모든 입력과 출력 데이터는 32-bit RGBA 형태의 텍스처 형태로 FBO를 생성하여 off-screen 렌더링에 사용되었다. OpenGL ES 2.0에서 shader 연산 결과는 반드시 한 장의 텍스처에 화소 단위로 저장이 되게 되는데 이는 알고리즘을 구현함에 있어서 커다란 제약사항이 된다. 이러한 제약 때문에 한 번의 렌더링에서는 4개의 disparity 후보에 대한 정합 결과만을 저장할 수 있다. 일반적으로 16개의 disparity 후보를 탐색하므로, 결국 동일한 렌더링을 4회 반복하여 수행하여야 하는 비경제적인 구조를 가진다. 이와 같이 shader 출력 데이터 제약사항에 따른 렌더링 수의 증가는 현재 embedded GPU를 이용한 병렬처리에서의 가장 큰 걸림돌이 되고 있다.



(a) (b)

그림 5. 스테레오 정합 결과 [3]
(a) Tsukuba 입력영상, (b) 생성된 Disparity 지도

PowerVR SGX 540에서 구현된 Belief Propagation 알고리즘의 수행 결과를 〈그림 5〉에 제시하였다. 또한 수행 속도 비교를 〈표 3〉에 제시하였다. Belief Propagation 알고리즘 내부의 다양한 내부 반복 횟수에 대해 CPU 대비 약 5~7배의 속도 향상을 확인할 수 있다.

표 3. 스테레오정합 알고리즘의 속도 비교 [3]
(단위: sec, CPU: ARM Cortex A8, GPU: PowerVR SGX 540)

| BP 알고리즘 내부 반복 수 | CPU | GPU | 가속비율 |
|---|---|---|---|
| 2 | 4.152 | 0.578 | ×7.18 |
| 4 | 6.976 | 1.086 | ×6.42 |
| 8 | 12.161 | 2.083 | ×5.84 |
| 12 | 17.413 | 3.125 | ×5.57 |
| 20 | 27.889 | 5.263 | ×5.23 |

## V. 결 론

본 기고를 통해 스마트폰의 응용프로세서가 탑재한 embedded GPU를 이용하여 영상처리를 수행하기 위한 하드웨어 기술동향, 소프트웨어 및 개발환경, 그리고 영상처리 응용 사례를 살펴보았다. 향후 관련 연구자들이 OpenGL ES 또는 OpenCL을 이용하여 많은 영상처리 응용 프로그램을 개발하고 스마트폰에서의 고속 영상처리가 더욱 활성화 되기를 기대하며 본 기고문을 마치도록 한다.

### Acknowledgement

본 연구는 인하대학교와 삼성전자의 지원에 의하여 수행되었음.

## 참 고 문 헌

[1] 권영수, 엄낙웅, "모바일 프로세서 기술 현황," 전자공학회지 제38권 5호, pp. 358-364, 2011년 5월.

[2] 이광엽, 박우찬, "모바일 그래픽 프로세서," 전자공학회지 제38권 5호, pp. 389-395, 2011년 5월.

[3] 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, September 2010.

[4] Khronos Group, OpenGL ES, http://www.khronos.org/opengles/.

[5] A. Munshi, D. Ginsburg, and D. Shreiner, OpenGL ES 2.0 Programming Guide (First Edition), Addison-Wesley, 2008.

[6] R.J. Rost and B. Licea-Kane, OpenGL Shading Language (Third Edition), Addison-Wesley, January 2009.

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

[8] 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, September 2006.

### 약 력

박 인 규

1995년 서울대학교 제어계측공학과 공학사
1997년 서울대학교 제어계측공학과 공학석사
2001년 서울대학교 전기컴퓨터공학부 공학박사
2001년~2004년 삼성종합기술원 멀티미디어랩 전문연구원
2007년~2008년 Mitsubishi Electric Research Laboratories (MERL) 방문연구원
2004년~현재 인하대학교 정보통신공학부 부교수
관심분야: 컴퓨터 그래픽스 및 비전 (영상기반 3차원 형상 모델링 및 렌더링, computational photography), GPGPU

최 호 열

2010년 인하대학교 정보통신공학부 공학사
2010년~현재 인하대학교 로봇공학전공 석사과정
2012년~현재 전자부품연구원 멀티미디어 IP센터 위촉연구원
관심분야: 대용량 병렬처리, 컴퓨터비전, 영상처리, 임베디드 병렬컴퓨팅