마이크로 프로세서 응용 실습

학번: 2019440121

이름: 조성원

목차

A. 목표

- 안드로이드 스튜디오 기반으로 Java 및 JNI, OPENCL Framework를 통한 어플리케이션 구현을 해본다.
- 카메라를 활용하여 이미지를 받아오고 해당 이미지에 대한 이미지 처리를 적용해본다.
- 원하는 연산에 대한 OPENCL Kernel 파일을 생성하여 GPGPU를 활용해본다.
- 현대 가속기 모델들을 파악하고 Precision에 따른 연산 속도를 비교해본다.

B. 기능

기능은 주기능과 부기능으로 나눌 수 있다. 주 기능은 실제 어플리케이션에 주된 기능이고 부기능은 디바이스 드라이버를 활용한 기능을 나타낸다.

주기능은

1. 카메라 촬영 및 캡처기능

2. 카메라를 통해 받은 이미지에 대해 CPU/GPU기반을 통한 Gaussian Blur 및 Gray scale 적용

3. Precision이 연산속도에 미치는 영향을 GPU를 통해 확인

4. Piezo의 주파수 영역대를 이용한 동요 연주

5. 버튼을 통한 안정적 어플리케이션 종료가 있다.

부가 기능은

Segment를 토대로 연산과정에 걸리는 시간을 나타내는 것과 버튼 순서에 따른 led 표시기능, Piezo 사용기능, 동요 연주의 박자에 따른 led 점멸 기능이 존재한다.

(Segment는 클럭에 따라 점멸하기에 한 프레임내 담기 힘들어 보고서에 사진을 담지 않았습니다.)

(하지만 시연에서 발표하였기에 그 시연 때 기억을 참고해주시면 감사하겠습니다.)

1) 카메라 촬영 및 캡처기능

카메라 촬영 및 캡처기능은 SurfaceView 클래스를 이용하여 실행해주었다.

카메라를 통해 이미지 frame을 받고 해당 이미지 frame을 Bitmap으로 decode를 해준다. 해당 하드웨어 사용한 카메라의 이미지는 기본 값이 90도의 회전이였기에 다시 180도 회전시켜주고 해당 이미지를 화면에 띄어준다.

캡처 버튼 실행시 SurfaceView에 있는 이미지를 가져와 scale down을 진행시킨다.

해당 scale down 작업을 통해 필요한 heap영역 메모리를 줄일 수 있다.

그다음 캡처된 이미지를 ImageView에 전달하여 화면에 띄어준다.

Fig. Application Interface

2) 카메라를 통해 받은 이미지에 대해 CPU/GPU 각각의 Gaussian Blur 필터 및 Gray Scale 적용

a. CPU/GPU 각각의 Gaussian Blur 필터 적용

카메라를 통해 위의 캡처 단계를 진행해준다.

그 다음 캡처된 이미지를 ImageView에 바로 보내는 것이 아니라 JNI내 코드에 있는 **CPU 및 GPU를 활용한 Filter 및 scaling**을 적용한다.

가우시안 블러는 Blur 처리를 하기 위한 필터로 필터의 크기에 따라 Blur의 강도가 결정된다.

Fig. Filter 크기에 따른 Blur 강도 비교

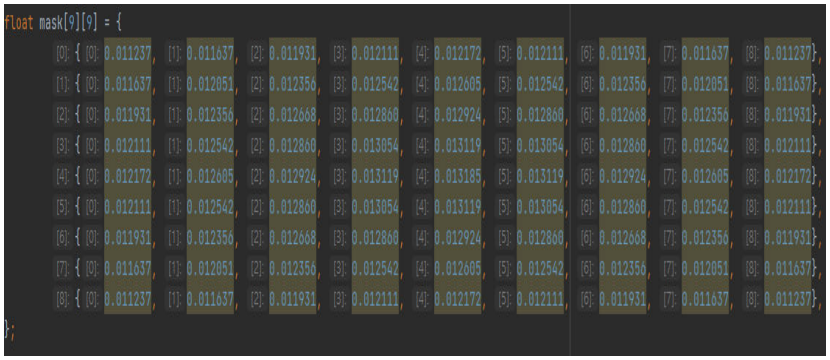


Fig. 실제 filter 값

해당 프로젝트는 **9x9 kernel**을 활용하였으며 해당 결과는 아래의 이미지와 동일하다.

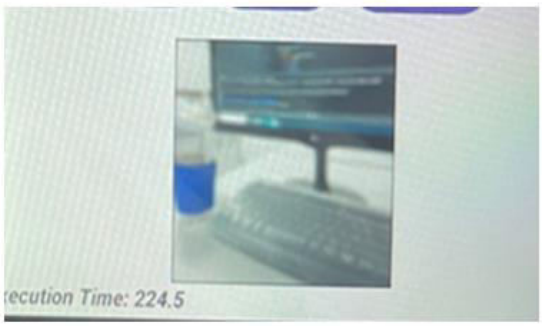


Fig. GPU를 통한 필터처리

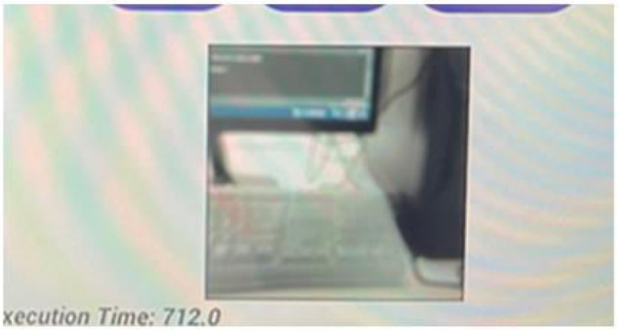


Fig. CPU를 통한 필터처리

위 결과를 살펴보면 **CPU를 통한 연산처리가 매우 늦다는 것**을 확인할 수 있다.

실제 컴퓨팅 성능을 확인하는데 사용되는 지표인 FLOPs를 기준으로 연산량을 나타낼 때 해당 **이미지의 input은 300x300x3 pixel**이고 **filter의 크기는 9x9**이다. 이를 토대로 연산량을 계산하면 약 **22MFLOPs**를 나타낸다. 또한 Filter를 통한 이미지 연산은 연산들이 서로 독립적으로 존재하기에 병렬처리하기 용이하며 이를 토대로 연산속도를 높일 수 있다.

GPU에 사용된 Blur.cl file은 아래와 같다.

```
#define RGB8888_A(p) ((p & (0xff<<24))      >> 24 )
#define RGB8888_B(p) ((p & (0xff << 16)) >> 16 )
#define RGB8888_G(p) ((p & (0xff << 8))  >> 8 )
#define RGB8888_R(p) (p & (0xff) )

__kernel void kernel_blur(__global int *src, __global int *dst, const int width, const int height){
    //int id = get_global_id(0);

    int row = get_global_id(0)/width;
    int col = get_global_id(0)%width;

    float mask[9][9] = {
        {0.011237, 0.011637, 0.011931, 0.012111, 0.012172, 0.012111, 0.011931, 0.011637, 0.011237},
        {0.011637, 0.012053, 0.012356, 0.012542, 0.012605, 0.012542, 0.012356, 0.012051, 0.011637},
        {0.011931, 0.012356, 0.012668, 0.012860, 0.012924, 0.012860, 0.012668, 0.012356, 0.011931},
        {0.012111, 0.012542, 0.012860, 0.013054, 0.013119, 0.013054, 0.012860, 0.012542, 0.012111},
        {0.012172, 0.012605, 0.012924, 0.013119, 0.013185, 0.013119, 0.012924, 0.012605, 0.012172},
        {0.012111, 0.012542, 0.012860, 0.013054, 0.013119, 0.013054, 0.012860, 0.012542, 0.012111},
        {0.011931, 0.012356, 0.012668, 0.012860, 0.012924, 0.012860, 0.012668, 0.012356, 0.011931},
        {0.011637, 0.012053, 0.012356, 0.012542, 0.012605, 0.012542, 0.012356, 0.012051, 0.011637},
        {0.011237, 0.011637, 0.011931, 0.012111, 0.012172, 0.012111, 0.011931, 0.011637, 0.011237},
    };

    int a,r, g, b;

    float red = 0, green = 0, blue = 0;
    int m, n,x,y;

    for (m = 0; m < 9; m++) {
        for (n = 0; n < 9; n++) {
            y = (row + m - 4);
            x = (col + n - 4);
            if ((row + m - 4) < 0 || y >= height || (col + n - 4) < 0 || x >= width) continue;
            int pixel = src[width * y + x];

            //
            a = RGB8888_A(pixel);
            r = RGB8888_R(pixel);
            g = RGB8888_G(pixel);
            b = RGB8888_B(pixel);


            red += r * mask[m][n];
            green += g * mask[m][n];
            blue += b * mask[m][n];
        }
    }
    r = (int) red;
    g = (int) green;
    b = (int) blue;

    int v = (a << 24) + (b << 16) + (g << 8) + (r);
    dst[width * row + col] = v;
}
```

Fig. Gaussian Blur를 위해 나타낸 Blur.cl

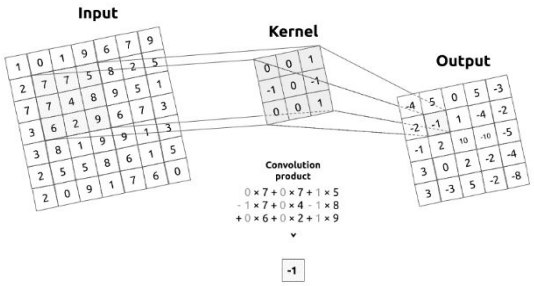


Fig. filter에 따른 2D-Convolution 연산

실제 연산과정을 살펴보면 테두리의 데이터에 대해선 필터연산이 적용되지 않음을 알 수 있다. 2-D Convolution연산은 Output data의 크기가 줄어들게 되는데 테두리 데이터에 대한 보존을 통해 Output data 크기의 감소가 일어나지 않는다.

그리하여 원래 사이즈의 Bmp file pixel data에 넣어 다시 bmp file로 만들고 해당 file을 ImageView에 넣어 화면에 띄운다.

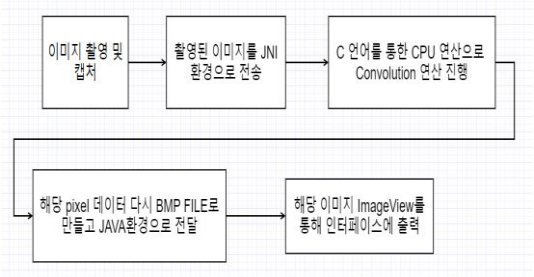


Fig. CPU를 활용한 Gaussian Blur 처리 과정

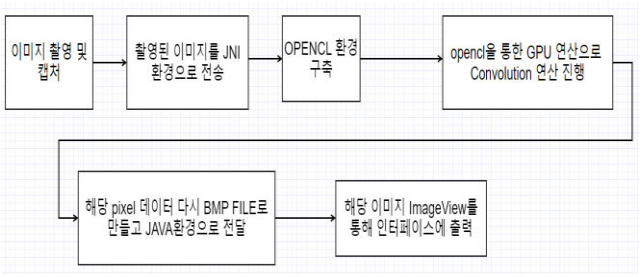


Fig. GPU를 활용한 Gaussian Blur 처리 과정

b. CPU/GPU 각각의 Gray Scaling 적용

Gray scaling은 일반적인 이미지 데이터 파일의 특징인 RGB 3개의 채널을 1개의 채널로 바꾸는 과정이다. 실제 해당 과정을 거치게 되면 이미지의 구조적 특징은 존재하지만 데이터의 크기는 그만큼 작아지게 된다.

(본 코드에서는 RGB 전체 채널에 동일한 데이터를 담기에 크기는 그대로가 된다. 하지만 RGB 채널을 한 개의 명암 채널로 합쳐도 상관없다.)

Gray Scaling의 데이터 연산량은 위의 Gaussian Blur의 연산량에 비해 매우 작다고 할 수 있다. **300x300x3 데이터에 대한 Gray Scaling 연산량은 0.36MFLOPs**에 해당하며 이는 위 연산량의 **약 1/60**이다. 이를 토대로 GPU, CPU 연산을 진행해보면 아래와 같이 나타난다.

Fig. GPU를 통한 scaling 처리

Fig. CPU를 이용한 scaling 처리

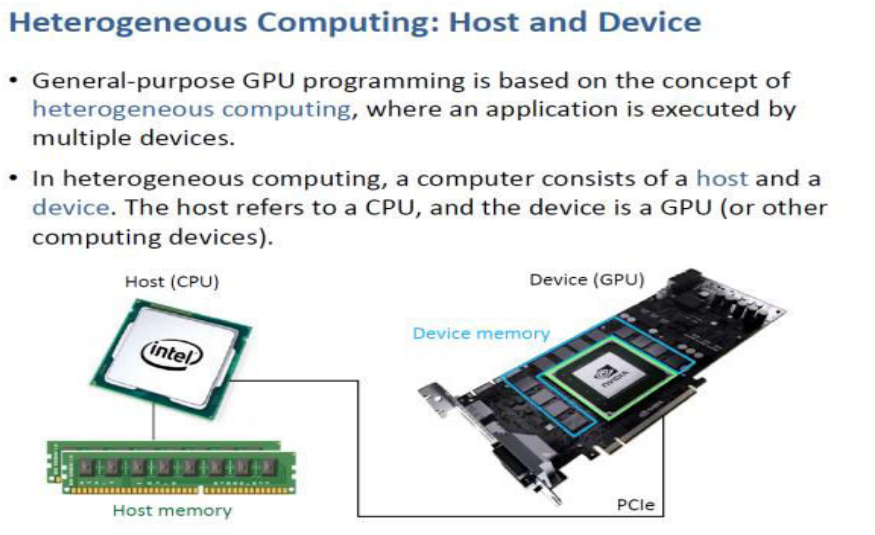


Fig. 이기종 디바이스를 통한 연산 시 시스템 개요

위와 같이 나타나는데 Gaussian Blur와 달리 GPU를 통한 연산처리가 CPU보다 느리다는 것을 확인할 수 있다. 실제로 OPENCL를 활용한 GPU 활용은 host memory (Main Memory)에서의 Device Memory(GPU memory)로의 데이터 이동이 존재한다. 이는 데이터 버스를 활용하기에 이동하는데 병목현상을 야기할 수 있고 실제 연산량이 적다면 CPU에서의 연산 시간보다 훨씬 느릴 수 있다. 위의 결과를 토대로 알 수 있다.

```
#define RGB8888_A(p) ((p & (0xff<<24))      >> 24 )
#define RGB8888_B(p) ((p & (0xff << 16)) >> 16 )
#define RGB8888_G(p) ((p & (0xff << 8))  >> 8 )
#define RGB8888_R(p) (p & (0xff) )

__kernel void kernel_gray(__global int *src, __global int *dst, const int width, const int height){
    //int id = get_global_id(0);

    int row = get_global_id(0)/width;
    int col = get_global_id(0)%width;

    int a,r, g, b;
    int gray;

    float red = 0, green = 0, blue = 0;
    int pixel = src[width * row + col];
    a = RGB8888_A(pixel);
    r = RGB8888_R(pixel);
    g = RGB8888_G(pixel);
    b = RGB8888_B(pixel);


    red = r * 0.2126;
    green = g * 0.7152;
    blue = b * 0.0722;
    gray = red+green+blue;

    int v = (a << 24) + (gray << 16) + (gray << 8) + (gray);
    dst[width * row + col] = v;
}
```

Fig. Gray Scaling을 위한 gray.cl 파일

```
pixel = ((uint32_t*)bitmapPixels) + y*info.width +x;
int r,g,b;
uint32_t v = *((uint32_t*)pixel);
b= RGB8888_B(v)*0.0722;
g= RGB8888_G(v)*0.7152;
r = RGB8888_R(v)*0.2126;
int gray = r+b+g;
int k = (a<<24) + (gray <<16 )+ (gray <<8) + (gray);
src[y*info.width+x] = k;
```

Fig. Gray Scaling 연산 과정

Gray scaling의 종류는 많이 존재하지만 교안에 나오는 방식을 이용하였다.

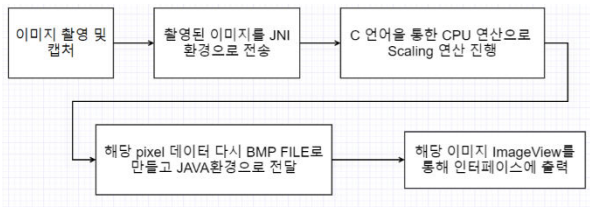


Fig. CPU를 활용한 Gray Scaling 처리 과정

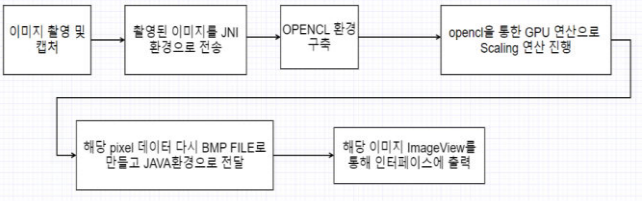


Fig. GPU를 활용한 Gray Scaling 처리 과정

위 방식은 CPU 방식 및 GPU 방식의 플로우 차트를 나타내며 동일한 연산과정을 통하기에 전체 연산량은 동일하다고 할 수 있다.

3) Precision이 연산속도에 미치는 영향 확인

인공지능의 시대는 하드웨어의 발전이 열었다고 해도 과언이 아니라는 말이 있다. 많은 데이터들을 처리하기 위해선 하드웨어 컴퓨팅 능력은 필수적이라 생각된다. 이러한 컴퓨팅 능력을 측정하는 기준에는 당연히 연산처리 시간이 존재한다.

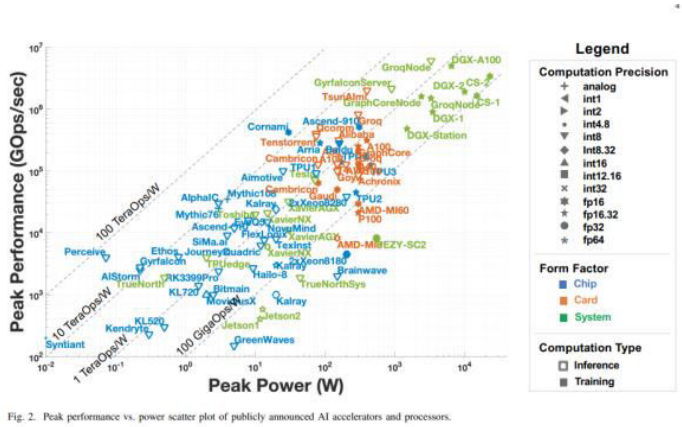


Fig. 2. Peak performance vs. power scatter plot of publicly announced AI accelerators and processors.

Fig. 현대 인공지능 모델 하드웨어의 성능 비교표

실제 위의 그림을 참고하면 많은 종류의 가속기 및 범용 모델 인공지능 하드웨어는 Low Precision을 가진다. 그 **이유는 인공지능의 모델은 98퍼센트의 확률과 95퍼센트의 확률 결과에 치명적으로 적용하지 않기 때문이다**. 그에 따라 인공지능 모델에 자주 사용되는 CNN 알고리즘을 기반으로 연산을 진행하였을 때 **low Precision이 효율적인지 확인하고자** 하였다.

기존의 BMP file 이미지를 읽어 해당 이미지에 하나의 커널(Filter)를 통과시켜 GPU내 걸리는 시간을 측정하는 것이다.

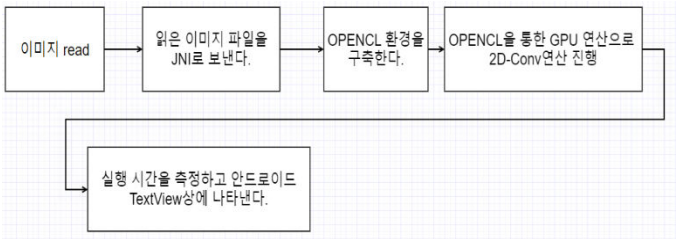
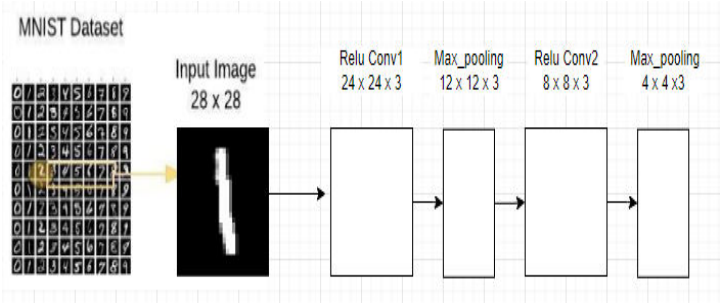


Fig. Precision에 따른 연산 속도 비교 과정

위 알고리즘을 토대로 진행된다. 평가를 하기위해 weight를 뽑아내고 해당 weight에 대해 Quantization을 진행해야한다.



Precision	FP64	FP32	INT8(weight only)	INT8
Acc	0.96	0.96	0.89	0.81
Loss	0.121	0.121	1.9	2.74

Fig. 간단한 filter 사용 (CNN)모델 구조, 정확도 및 양자화 후 모델 정확도

위 간단한 모델은 **Convolution Layer의 가중치를 뽑아내고 아래의 양자화 기법을 토대로 INT8** 형태로 양자화를 진행 후 성능을 확인한 것이다. 양자화에 대한 효용성은 이미 많은 곳에서 입증되었으며 Tensorflow, Torch 등 Machine Learning framework에서는 이미 양자화에 대한 많은 기능을 제공한다.

$$s = \frac{2^b - 1}{\alpha - \beta} \qquad \text{clip}(x, l, u) \begin{cases} l, & x < l \\ x, & l \le x \le u \\ u, & x > u \end{cases}$$

$$z = -\text{round}(\beta \cdot s) - 2^{b-1} \qquad x_q = \text{quantize}(x, b, s, z) = \text{clip}(\text{round}(s \cdot x + z), -2^{b-1}, 2^{b-1} - 1)$$

Fig. Affine Quanitzation

Fig. 9x9 Kernel float (float32)

Fig. 9x9 Kernel quantization (INT8)

실행결과를 보면 둘 다 Opencl을 통해 GPGPU기반으로 연산되었지만 연산의 속도는 서로 매우 다름을 알 수 있다.

Precision이 float32인 경우는 precision이 INT8인 경우보다 매우 높음을 알 수 있다.

(위 사용된 Affine Quantization은 integer에 대한 양자화에 많이 사용되는 구조라 할 수 있다.)

4) Piezo의 주파수 영역대를 이용한 동요 연주

Piezo 디바이스 드라이버를 활용하면 피에조를 연주할 수 있다. 실제 피에조 드라이버내 각각의 주파수 대역대별 나타낼 수 있는 음이 지정되어 있다.

디바이스 드라이버는 해당 주파수 영역을 7개의 피아노 건반과 맞춰 해당 연주를 진행할 수 있도록 만들어져 있었다.

디바이스 드라이버는 **file descriptor**로 관리되기에 **led, piezo, segment**를 사용하기 위해선 각각 서로 다른 파일 번호를 지정해주어야 한다.

(각각의 상세한 설명은 코드 파일에 주석을 통해 설명하였습니다.)

5) 버튼을 이용한 어플리케이션 종료

해당 프로젝트 어플리케이션은 **이미지 사진을 통해 많은 데이터 사용량**을 지니게 된다. 이러한 것이 해당 하드웨어에 무리를 줘 프로세스들이 예상보다 빠르게 종료되는 상황을 야기시켰다. 이는 하나의 캡처된 이미지만 사용하는 것이 아니라 blur, gray, cpu, gpu 마다 새로 캡처된 이미지를 활용하기 때문이라 생각된다.

만약 버튼 blurcpu, blur_gpu, gray_cpu, gray_gpu 4개의 버튼을 누른다면 4번의 캡처가 이뤄지고 4개의 이미지가 메모리에 생성되게 된다는 것이다.

이러한 문제를 해결하기 위해 어플리케이션의 안정적 종료 버튼을 구현하였다.

(각각의 상세한 설명은 코드 파일에 주석을 통해 설명하였습니다.)

결론 및 느낀점

- 자바를 이용하여 안드로이드 어플을 만드는 것은 처음이었고 OpenCl framework 또한 처음 사용해보아서 구현과정에서 많은 어려움을 겪었다.
- 또한 자바내 Thread가 어떻게 동작하는 지 깊게 이해하지 못해 구현은 해내도 왜 동작하는 지 의문이 들기도 하였다.
- 연산과정의 흐름을 커널로 구현하여 GPGPU를 사용할 수 있음을 알았고 앞으로 진로에서 많은 도움될 거 같다.
- Low precision에 대해 성능은 알았지만 하드웨어적 환경이 없어 직접 어느정도인지 체감한 적은 없다. 9x9커널을 사용하는 인공지능 모델은 드물지만 해당 연산을 통해 얼마나 Low precision이 시간을 덜 잡아먹는 지 확인하였다.

(아래에 조교님 질문에 대한 답변이 존재합니다.)

(부록. 조교님의 질문에 대한 답변)

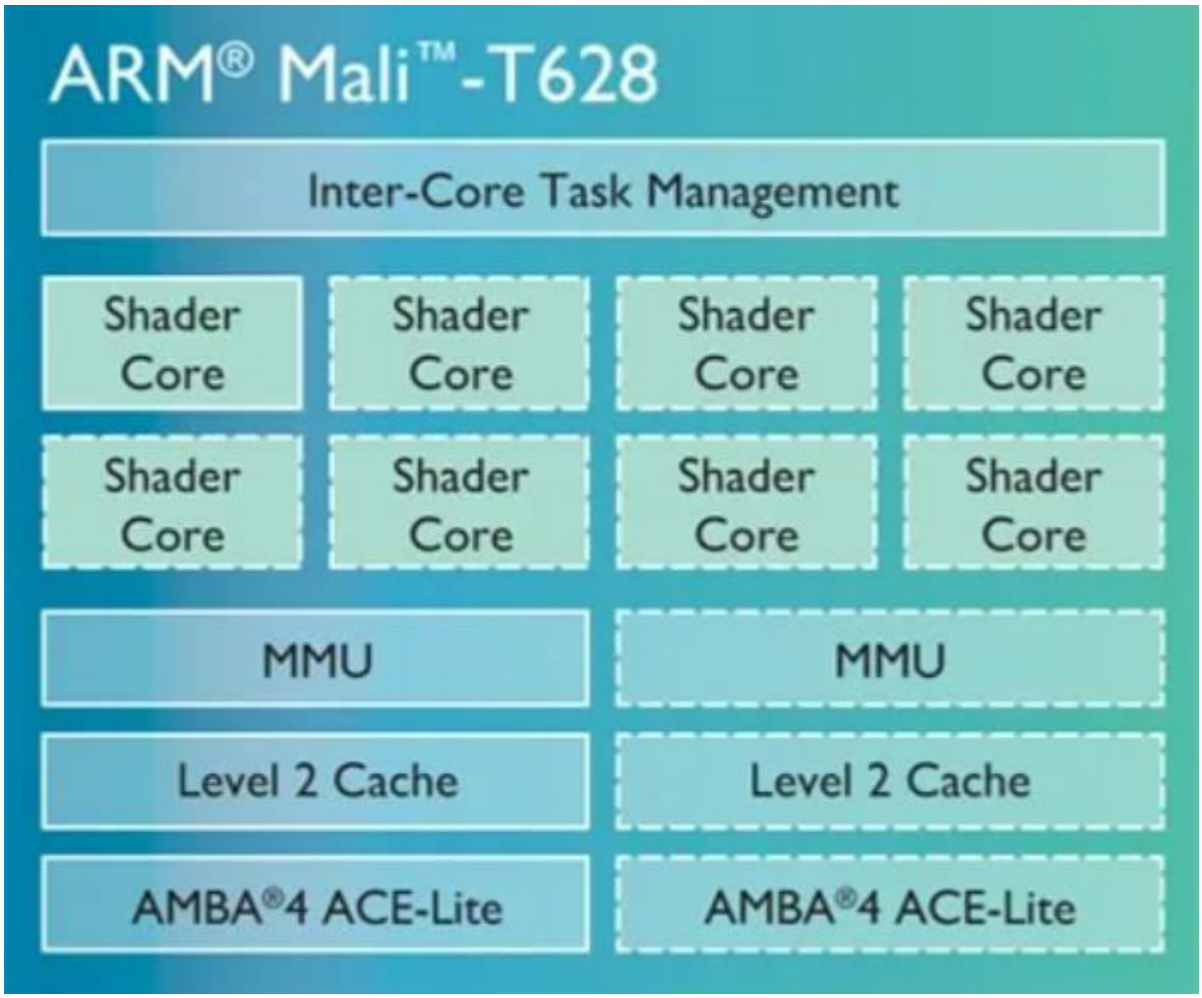


Fig. 본 프로젝트 하드웨어에 사용되는 GPU 구조

질문)

보통의 gpu에는 정수기반 연산기와 fpu를 따로 두지않고 FPU위주로 두어 연산이 진행된다. 그렇게 된다면 INT8로 변환시켜도 연산 속도 시간의 차이가 크지 않을 것인데 해당 프로젝트에서는 연산시간 차이가 존재하였다. 이는 왜 그런 것인가?

답변)

조교님 말씀대로 fpu위주로 연산기가 설정된 gpu들도 존재한다.

하지만 해당 프로젝트 하드웨어의 gpu 구조(Mali-T628)를 살펴보면 ALU연산기와 FPU 연산기가 설정된 Shader Core가 존재한다. 이러한 코어 구조에서는 Float 연산과 integer 연산에 따라 유연하게 서로 진행할 수 있다. 만약 FPU만 존재하는 GPU에서는 int8로 연산이나 float32의 연산이 동일하게 느껴질 것이다. 하지만 해당 프로젝트 gpu는 정수전용 연산기와 FPU가 따로 존재하기에 서로의 연산시간이 많이 차이남을 확인할 수 있었다.