ㅇㅇㅇ

[소프트웨어 종합설계]

Runtime Estimation for Multi-Core Processor

Case study of Exynos 4412

ㅇㅇㅇ

조장 ㅇㅇㅇ

조원 ㅇㅇㅇㅇㅇㅇ

담당 교수 ㅇㅇㅇ교수님

ㅇㅇㅇ Laboratory

ㅇㅇㅇ

목차

ㅇㅇㅇ

1. 목표

Multi-Core CPU와 GPU의 Power Modeling을 통한 실시간 전력 측정

2. 연구 내용

2.1 Multi-Core CPU

2.1.1 사전 조사

DVFS (Dynamic Voltage and Frequency Scaling)

디바이스에서 수행되는 작업들 모두가 최고의 CPU의 성능을 요구하는 것은 아니다. 또한 사용자의 입장에서 더 빠른 작업속도보다 더 긴 배터리 지속시간을 원하는 경우도 있다. 이러한 인식에 기초하여, 부품의 동작속도를 줄이는 대신 소비전력에서 이득을 취하고자 하는 기술이 DVFS이다.

$$E_{dynamic} = C \cdot f \cdot V^{2 \cdot t} \text{ (C는 상수)}$$

$$V \propto f,\ t \propto \frac{1}{f}$$

위의 식에서 f는 주파수이고 V는 소자에 공급되는 전압이며 t는 동작된 시간이다. 위의 관계를 통해 CPU에서 소모되는 에너지와 전압 간에는 다음과 같은 관계식이 성립한다.

$$\text{CPU에서 소비되는 에너지 } E \propto f^2$$

따라서 frequency를 조금만 줄이더라도 많은 에너지를 줄일 수 있기 때문에 DVFS는 효율적인 에너지 절감 기법이라고 할 수 있다.

Exynos4412에서는 아래 화면과 같이 코어마다 0.2GHz~ 1.4GHz까지 13개의 frequency를 제공하고 있다. cpu0에서 frequency를 변경하면 나머지 코어도 함께 변한다.

```
root@android:/sys/devices/system/cpu/cpu0/cpufreq/stats # cat time_in_state
cat time_in_state
1400000 34196
1300000 1011
1200000 2746
1100000 3214
1000000 9348
900000 1071
800000 27049
700000 2006
600000 33533
500000 48218
400000 0
300000 0
200000 2099462
```

CPU Governor

CPU 사용방법에 대한 정책으로 CPU의 frequency를 조절하는 방식이다. 갤럭시S3의 경우는 아래와 같이 8가지의 CPU Governor가 존재하고 cpu0의 governor를 조절하면 나머지 코어들도 같은 governor로 동작한다. 일반적인 스마트폰에서 사용되는 governor는 ondemand로 cpu의 과부에 따라 frequency를 조절한다. 갤럭시S3의 경우는 기본적으로 pegasusq를 사용한다. pegasusq란 ondemand를 기반으로 hotplug의 요소를 집어넣은 갤럭시S3 전용 governor이다. Hotpluging을 통해 사용하지 않는 cpu의 전원을 차단해서 전력 소비를 차감한다. 우리는 모델링 실험을 위해서 지정해준 frequency를 고정해서 사용하는 userspace를 사용하였다.

```
root@android:/sys/devices/system/cpu/cpu0/cpufreq # cat scaling_available_governors
vailable_governors                                                              <
adaptive interactive conservative ondemand userspace powersave pegasusq performance
```

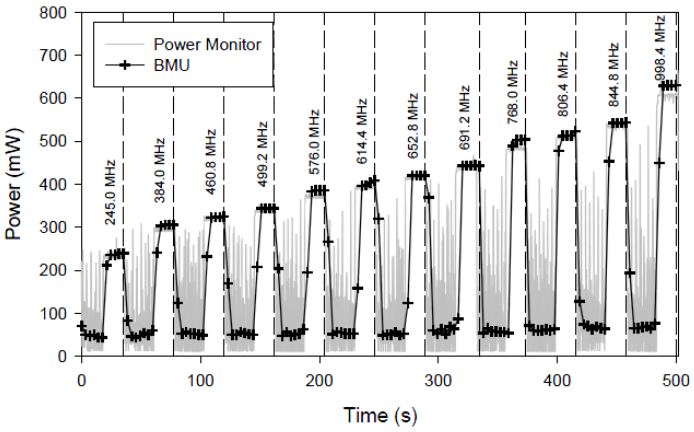
Idle governor

갤럭시S3의 특이점 중 하나는 cpu0에 idle state가 2가지가 존재한다는 것이다. 다른 cpu1~3의 경우는 idle state가 IDLE만 존재하는데 cpu0의 경우는 IDLE과 LOW_POWER가 존재한다. LOW_POWER의 경우는 전력을 frequecy에 영향을 거의 받지 않고 아주 낮게 소모한다. 보통 스마트폰을 거의 사용하지 않을 경우 전력 소비를 절감하기 위해 LOW_POWER로 cpu0만 켜진 상태를 유지한다.

```
root@android:/sys/devices/system/cpu/cpu0/cpuidle # ls
ls
state0
state1
root@android:/sys/devices/system/cpu/cpu0/cpuidle # cd state0
cd state0
root@android:/sys/devices/system/cpu/cpu0/cpuidle/state0 # cat name
cat name
IDLE
root@android:/sys/devices/system/cpu/cpu0/cpuidle/state0 # cd ..
cd ..
root@android:/sys/devices/system/cpu/cpu0/cpuidle # cd state1
cd state1
root@android:/sys/devices/system/cpu/cpu0/cpuidle/state1 # cat name
cat name
LOW_POWER
root@android:/sys/devices/system/cpu/cpu0/cpuidle/state1 #
```

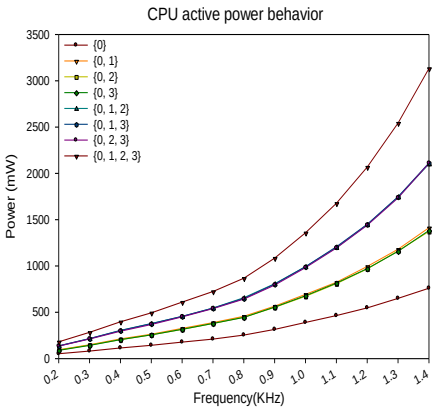
2.1.2 Utilization 기반 모델링

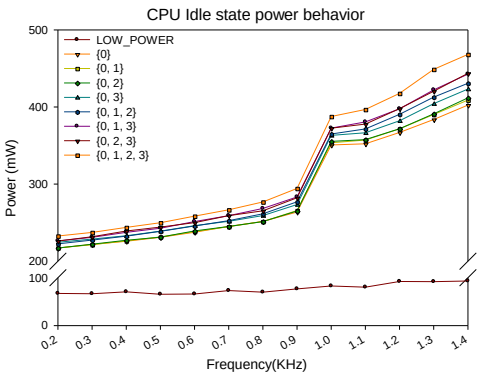
$$P^{CPU} = \beta_{freq}^{CPU} \times u + \beta_{freq}^{idle}$$
u: utilization, $0 \le u \le 100$
$freq$: frequency index, $freq = 0,1,2 \cdots, n$

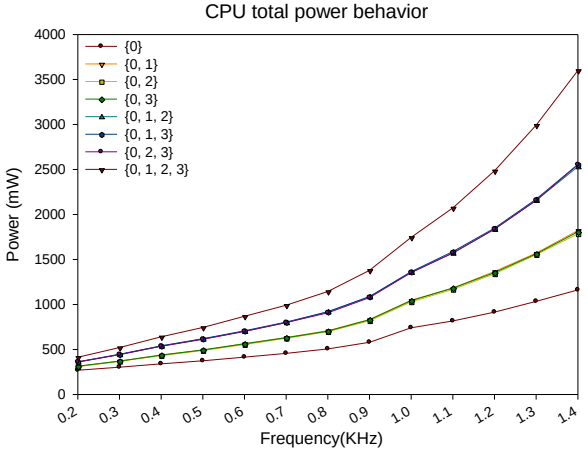


이전 모델링은 CPU가 기본적으로 사용하는 idle state의 소비 전력과 Utilization에 따라 소비되는 active state의 전력 값의 합으로 이루어진다. Utilization이 0%일 때 전력 값을 측정해 idle state의 coefficient값으로 설정하고 utilization이 100%일 때 active state의 coefficient 값으로 설정해서 위의 식에 적용한다. 각각의 frequency에 따라 이 coefficient값은 차이가 난다. 이 모델을 바탕으로 Multi-Core에 대한 전력을 측정해보았다.

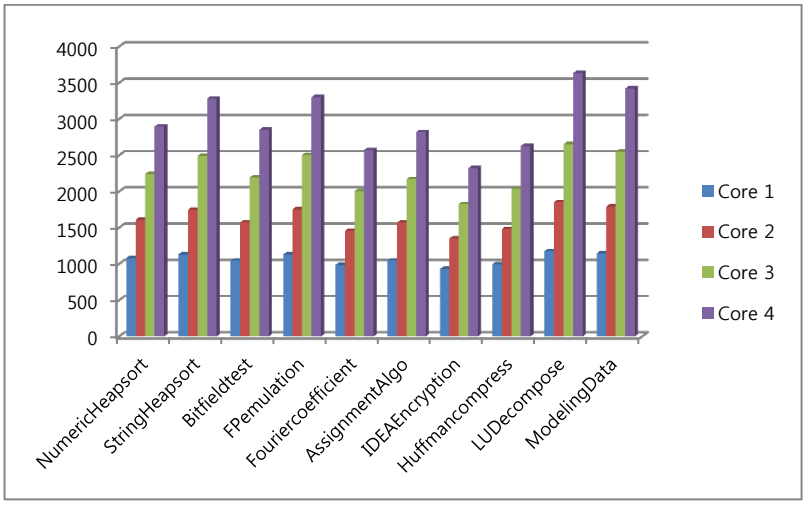
$$P_{CPU} = \sum_{j=0}^{frequencies} \left(\beta_{i,j}^{freq} * u_j^{freq} + \sum_{k=0}^{idle\ levels} \beta_{i,j,k}^{idle} * u_{j,k}^{idle} \right),$$
$$i = num\ of\ active\ core$$







이 실험을 통해서 전력은 켜진 코어의 번호와는 관련이 없고 켜진 개수만 관련이 있다는 것을 확인했고 Utilization 모델이 멀티코어에도 잘 적용되는 것을 확인할 수 있었다. 하지만 이것은 같은 종류의 연산의 경우만 적용이 가능하다는 점에서 문제가 있다.



Nbench의 각 연산의 소비 전력을 측정해봤다. 모두 Utilization은 100%로 동일하지만 소비 전력은 차이가 났고 이 차이는 코어가 1개 일때는 245.12mW 정도인데 코어가 4개일 때는 1308.27mW로 큰 차이를 보인다. 그러므로 Utilization 모델을 Multi-Core CPU에 적용시키는 것은 큰 오차를 발생시킬 우려가 있다.

ㅇㅇㅇ

2.1.3 HPC 기반 모델링

About HPC

HPC, Hardware Performance Counter는 하드웨어의 성능을 측정하기 위한 회로로, 지원하는 이벤트들의 발생횟수를 측정하는 방법으로 성능을 알 수 있게 해주는 기능을 한다. 본 연구에서 사용하는 HPC는 Galaxy S3 의 AP인 Exynos4412의 내부에 들어가있는 HPC를 이용할 것이다.

HPC approach

- PERF

리눅스 커널소스에 기본적으로 탑재되어 있는 Performance monitoring tool이다. 리눅스와 마찬가지로 오픈소스 소프트웨어로 ARM아키텍쳐를 위한 버전도 준비되어있다. 동작방식은, HPC 값을 기반으로 성능을 측정하며, hpc값에 접근할 수 없는 경우에도 커널에서 제공하는 다양한 정보를 통해 성능에 관련된 다양한 정보를 얻는다.

하지만, PERF를 통해서는 실시간으로 CPU의 동작정보를 알기 힘들고, 또한 단순한 HPC값을 가져오는 기능외에도, 다른 기능들이 들어가 있기 때문에 OVERHEAD가 크고, 실험에 사용하기 힘든 단점이 존재한다.

- Direct approach

HPC는 PERF와 같은 툴을 통해서도 접근할 수 있지만, ARM에서는 HPC를 이용할 수 있도록 인라인 어셈블리 문법을 제공하여 직접 접근 또한 가능하다. 실험을 위해서 HPC값을 유저모드에서 읽어올 수 있도록 해주는 커널 모듈을 작성하였고, NDK를 이용하여 최종적으로는 유저 어플리케이션 수준에서 HPC값을 읽을 수 있도록 하였다.

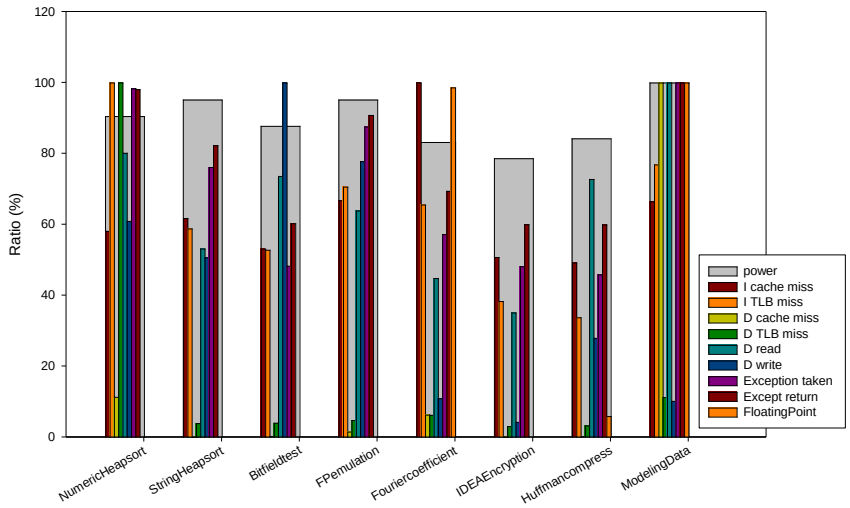
HPC Data

Exynos4412 내부의 HPC는 커널코드 상에서는 PMU(Performance Monitoring Unit)으로 불리며, Exynos4412의 베이스인 ARM Cortex A9 아키텍쳐의 PMU에서 지원하는 총 61가지 이벤트를 측정할 수 있게 설계되어 있다. 이 이벤트들은 Instruction cache miss, Data read/write, Floating point unit operation 등 cpu에서 수행되는 대부분의 기능들을 측정할 수 있도록 설계되어있다.

하지만, HPC는 1개의 cycle register와 6개의 counter register로 이루어져, 동시에 측정할 수 있는 이벤트의 수가 최대 6개로 제한되어 있다.

HPC Data Analysis

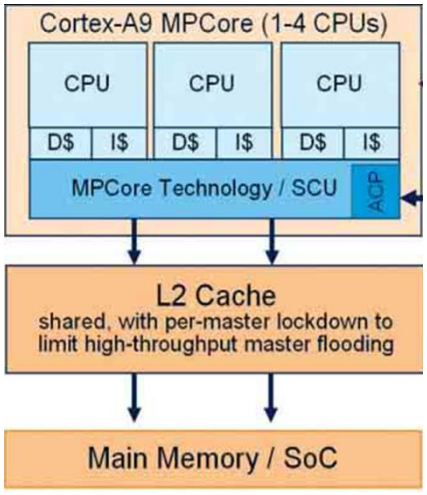
동시에 측정 가능한 이벤트의 수가 6개로 제한됨에 따라서, Power Modeling에 사용할 변수로서의 이벤트의 종류를 결정해야 할 필요성이 존재한다. 따라서 먼저 61가지의 이벤트중 cpu의 동작 및 반복횟수에 관련된 이벤트들을 선정하여 벤치마크를 수행하는 동안 hpc값의 결과를 얻었다.



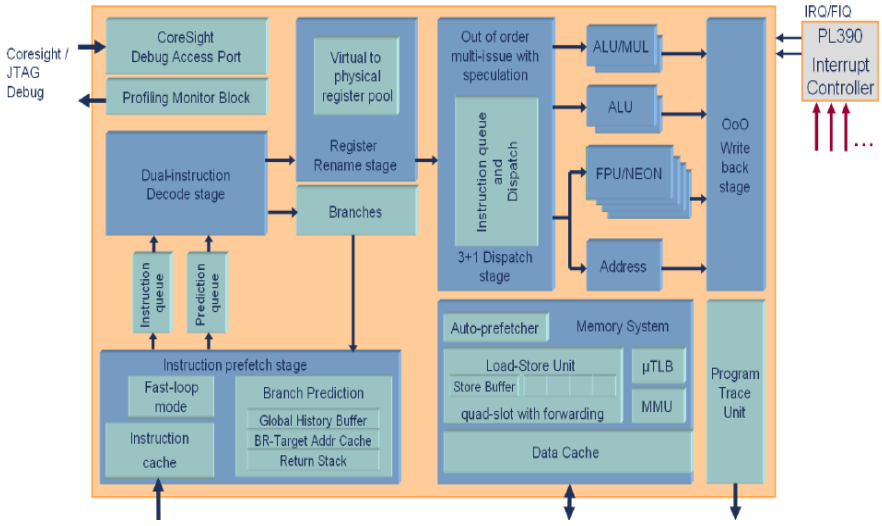
위 그래프는 해당 실험의 결과이다. 위 그래프에서도 알 수 있듯이 소비전력은, 적은 수의 이벤트에 의해 지배적으로 영향 받지 않으며, 좀 더 많은 수의 변수에 의해 결정될 것이라 예측된다.

Architectural Analysis & HPC modeling

이전까지의 결과를 보면 단순한 회귀분석을 통한 Power model을 만드는 것보다, 구조적인 분석을 통한 접근이 필요하다고 생각된다. 아래의 블록 다이어그램은 Exynos4412 플랫폼을 단순화한 그림이다. 크게 CPU core, L1cache(Data, Inst), L2cache, Memroy로 구분할 수 있다.



CPU core



위 그림은 ARM Cortex-A9의 블록다이어그램이다. 다이어그램에서는 포함되지 않았지만 PMU(Performance Monitoring Unit)가 코어내에 존재하여, HPC 값을 계산하며, CPU코어의 사용전력에 관계될 것이라고 추측되는 HPC 값들은 다음과 같다.

- NumInstruction : Core에서 실행된 명령어의 개수로, Decode가 종료된 시점에서 측정된다.

- (Main/Second/FPU/NEON/LoadStore/Branch) Unit Instrcution : 각 명령어가 처리되는 유닛에서 처리된 명령어의 수를 각각 측정할 수 있다.

- Clock cycle : Cpu에서 동작과정이 수행된 clock의 수 이다. 이 Clock cycle은 실제 수행된 명령어의 수에 비례하며, 최대로 동작하고 있을경우, cpu의 동작 frequency에 비례한다. 따라서 Cpu의 frequency와 utilization에 비례하는 수치이다.

$$P_{CPU\ core} = \beta_{inst} \times NumInstruction + \beta_{clock} \times (CPUClock)^2$$

β_{inst}와 β_{clock}은 linear regression을 통해 얻은 coefficient이다. ARM은 32-bit RISC 구조이므로 모든 명령어가 고정된 길이를 가진다. 그러므로 명령어 유닛마다 처리된 명령어 수가 아닌 총 명령어를 사용하는 것이 큰 문제가 없을 것으로 판단되었다. 그리고 CPU Clock의 제곱을 사용하는 이유는 위에 DVFS 부분에 설명했듯이 CPU에서 소비되는 에너지가 frequency에 비례하기 때문이다. 그리고 frequency와 CPU Clock이 비례하기 때문에 결론적으로 전력은 CPU Clock의 제곱에 비례하게 된다. 그래서 CPU core에서 사용하는 전력의 모델링은 위의 식과 같다.

Cache

Exynos4412 아키텍쳐에서는 각 코어별 L1cache와 모든 코어가 공유하는 L2cache의 두단계로 Cache를 이용한다. L1cache와 L2cache를 구분해서 전력소모를 계산하지 않는 것은, L1과 L2 cache가 밀접하게 연관되어 있기 때문에 HPC값으로 각 소모전력을 분산해서 계산하기 어렵기 때문이다. Cache와 관련된 HPC 값들은 다음과 같다.

- Data Cache (Access/Miss) : Data cache에 Access한 횟수와, Miss가 발생한 경우이다. Access의 경우에는 L1 cache에서만 Load가 발생하지만 Miss의 경우에는 L1, L2 양 쪽에서 모두 Load가 발생한다.

- Instruction Cache (Access/Miss) : Data Cache (Access/Miss)와 흡사하며, 차이점은 Data를 가져오는 것이 아닌 명령어가 저장된 페이지에 접근하는 것으로 대부분의 프로그램에서는 Caching효율이 매우 좋아 miss의 비율이 적으며, Data cache에 비해 적은 전력을 소모한다.

- Data (Read/Write) : 명령어가 실행되는 과정에서 데이터를 읽은 횟수와, 데이터를 쓴 횟수이다. Data Read/Write와 Data cache access와는 유의하게 사용되어야 하는 이유는 데이터를 읽는데 드는 overhead보다 쓰는데 있어 overhead가 크기 때문이며, 이것은 멀티코어 cpu에서는 코어별로 L1cache를 가지지만 공통의 L2cache를 가진다는 점에서 더욱 유의하게 나타난다.

$$P_{Cache} = \beta_{read} \times \frac{DReadInst}{LD\&ST\ Inst} \times DCacheAccess + \beta_{write} \times \frac{DWriteInst}{LD\&ST\ Inst} \times DCacheAccess + \beta_{miss} \times DCacheMiss$$

β_{read}와 β_{write} 그리고 β_{miss}는 linear regression을 통해 얻어진 coefficeint이다. Load & Store Instruction은 Data Read Instruction과 Data Write Instruction을 합한 것과 거의 일치하는 값을 가진다. Data Read Instruction / Load & Store Instruction은 데이터를 사용할 때 데이터를 읽어온 비율을 나타내고 Data Write Instruction / Load & Store Instruction은 데이터를 사용할 때 데이터를 쓴 비율을 나타낸다. 이 비율을 Data Cache Access에 곱해서 Cache Access 중에 Read와 Write 부분을 구별한다. 마지막으로 Data Cache Miss 부분도 전력에 영향을 미칠 것으로 예상하고 Cache의 전력 소비 모델링을 위와 같이 정했다. Cache에 관한 부분은 아직 모델링 부분이 많이 변형되고 있고 그것을 뒷받침할 자료를 정리 중에 있다.

Memory

Memory는 독자적인 작동이 불가능 하고, Bus를 통해 저장된 정보를 전송하거나, 새로운 정보를 전송받아 저장하는 역할을 수행하기 때문에 실질적인 메모리의 사용은 Bus의 사용량과 직결된다. 따라서 우리는 Memory의 사용전력을 추산하는데 Bus의 사용량을 사용하였다. Galaxy S3에서는 PPMU(Platform Performance Monitoring Unit)이 존재하여 플랫폼에서의 HPC값을 읽을 수 있으며, 지원되는 데이터중 Bus와 관계된 것은 다음과 같다.

Bus Clock : Bus의 동작클럭으로 커널에서 동적으로 조작이 가능한 Bus_freq에 따라서 고정된 값을 가진다. Bus클럭이 높을수록 데이터 전송속도가 빨라지나 소비전력이 늘어나는 Trade Off를 가진다.

BusTransaction : 버스의 사용 횟수를 나타내며, Bus Clock으로 나누어 초당 사용빈도를 계산할 수 있다.

$$P_{Bus} = \beta_{memory} \times BusTransaction + \beta_{Bclock} \times BusClock$$

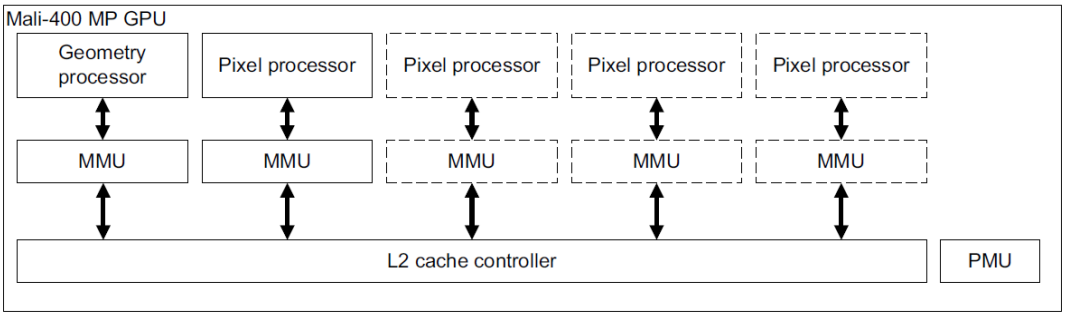
$\beta_{transaction}$와 β_{Bclock}은 linear regression을 통해 얻은 coefficient이다. BusTransaction은 버스의 사용량을 반영하고 BusClock은 버스의 frequency를 반영한다. 따라서 Memory의 전력 소비 모델링은 위의 식과 같다.

2.2 Graphic Processing Unit

2.2.1 Mali-400MP

Mali-400MP는 Exynos 4412에 채택된 ARM사의 GPU이다. ARM의 Mali GPU는 이와 같이 자사의 CPU와 함께 사용되는 경우가 많으며, 모바일 기기나 임베디드 시스템에서 타일 기반의 렌더링 처리를 담당한다. 이 Mali-400MP는 Exynos 4412의 Cortex-A9 CPU와 마찬가지로 코어가 여러 개인 multi-core 형태이며, CPU는 모든 코어가 동일한 homogeneous multi-core인데 비해 GPU는 1개의 Geometry Processor(Vertex Processor)와 4개의 Pixel Processor(Fragment Processor)로 이루어져 있다. 이 외에도MMU(Memory Management Units)와, PMU(Power Management Unit), 그리고 L2 cache도 함께 Mali-400MP를 구성하고 있다.

주요 구성 요소들에 대해서 살펴보면, Geometry Processor는 vertex 생성 및 처리를 담당하며 primitive list, polygon list, packed vertex data를 생성하여 Pixel Processor에게 넘겨주는 역할을 한다. Pixel Processor들은 fragment 처리를 담당하며 Geometry Processor에서 생성한 list들과 여러 data structure들을 활용하여 이미지를 생성하는 작업을 담당한다. 마지막으로 L2 cache는 일반적인 cache 기능과 마찬가지로 memory bandwidth usage와 power consumption을 줄이는 역할을 하며 특히 memory에 접근하는 cost를 줄이기 위한 것이 주목적이다.



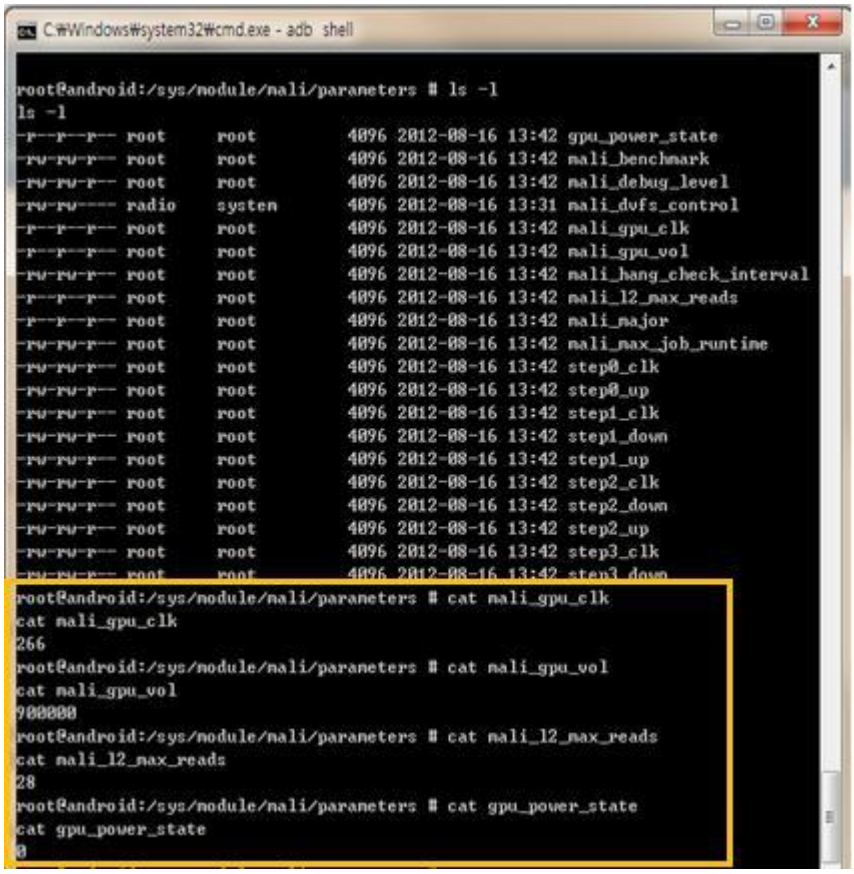
2.2.2 Mali GPU Utilization

GPU의 전력 소비를 모델링하기 위해서는 기본적으로 GPU만의 소비 전력은 물론 GPU가 언제, 얼마나 쓰이는지에 대한 정보들이 반드시 필요하다. 이 정보는 크게 GPU의 Utilization에 기반하는 데이터와 앞서 CPU 부분에서 설명한 HPC(Hardware Performance Counter)에 기반 하는 데이터로 나뉘는데, HPC 기반의 데이터로 모델링하는 것이 GPU에서도 보다 정확한 결과를 가져다 줄 것으로 예상되나 대상 기기에서는 GPU 관련 HPC를 사용하는 것이 실질적으로 불가능하여(HPC로 하여금 GPU관련 정보를 수집하도록 커널을 수정하였을 경우 기기가 부팅되지 않는 문제 발생) Utilization 기반으로 모델링을 하기로 결정하였다. CPU의 경우는 이미 Utilization 기반의 모델링이 존재하여 이를 HPC 기반으로 개선하는데 의미를 두었다면, GPU는

이전에 스마트폰 GPU에 대한 제대로 된 모델링이 존재하지 않았으므로 Utilization 기반의 모델링이라도 그 연구에 의의가 있다 할 수 있겠다.

Mali GPU를 사용하는 안드로이드 스마트폰 상에는(/sys/module/mali/parameters/) 아래 이미지와 같은 파일들이 존재한다. 이는 커널 옵션 중 'mali profiling'에 의해 생성된 것으로 실제 커널 코드 상에도 이와 관련한 소스들이 존재한다. 이 parameter들은 Mali GPU와 관련된 여러 데이터들을 각각 가지고 있다. step0부터 step3로 시작하는 파일들은 Mali GPU의 DVFS(Dynamic Voltage and Frequency Scaling)에 관련된 것인데, Mali-400MP의 경우 총 4개의 step이 존재한다. 각 step은 서로 다른 frequecny 및 voltage를 가지는데, Exynos 4412의 경우 이 4개의 step을 활용하여 각 상황마다 가장 적은 전력 소모로 작업을 처리할 수 있는 최적의 step을 선택한다.

참고로 각 step의 frequency를 살펴보자면, step0 : 160Mhz, step1 : 260Mhz, step2 : 350Mhz, step3 : 440Mhz이고, 기본 대기화면인 런쳐나 일반적으로 사용하는 어플리케이션의 경우 GPU를 사용하긴 하지만 GPU가 처리해야할 양이 적으므로 상대적으로 낮은 클럭에서 동작한다(step0, 1). 한편, GPU의 성능을 한계까지 시험하는 GPU benchmark 어플리케이션(GPU-T, NenaMark1, NenaMark2, BaseMark GUI Free, BaseMark Taiji Edition 등)이나 3D 그래픽이 많이 포함된 고사양의 게임 어플리케이션을 실행하는 경우에는 높은 동작 클럭(step2, step3)를 사용하는 것을 mali parameter 중 'mali_gpu_clk'를 통해 확인할 수 있었다. step으로 시작하는 것을 제외한 mali paramter들은 모두 실시간으로 바뀌는 데이터들이므로 필요할 때 유효한 데이터로 사용할 수 있다.



이전 연구(Appscope)의 경우 모든 하드웨어 컴포넌트들의 모델링이 Utilization 기반으로 이루어졌고, 사용 시간 등 다양한 정보를 바탕으로 각각의 Utilization을 직접 구해 모델링에 사용하였

다. 따라서 이번에 Utilization 기반의 GPU 모델링을 하게 되면서 직접 Utilization을 마찬가지로 직접 계산하려 했으나, 커널 소스 중 Mali 관련한 경로들을 모두 확인한 끝에 Utilization을 계산하는 함수를 찾아 이를 활용하게 되었다. Mali GPU의 Utilization을 계산하는 이 함수는 말 그대로 'calculate_gpu_utilization'이라는 이름을 가지고 있으며 drivers/media/video/samsung/mali/common/mali_kernel_utilization.c 라는 경로에 위치해 있다. calculate_gpu_utilization 함수는 커널 상에 정해진 기준 시간(1000ms = 1초, 수정 가능)마다 GPU의 현재 Utilization을 계산해 전용 핸들러인 mali_gpu_utilization_handler에 계산된 Utilization 값을 넘겨준다. 참고로 이 Utilization 값은 0 이상 256 이하의 범위를 가진다.

```
93      utilization = work_normalized / period_normalized;
94
95      accumulated_work_time = 0;
96      period_start_time = time_now; /* starting a new period */
97
98      _mali_osk_lock_signal(time_data_lock, _MALI_OSK_LOCKMODE_RW);
99
100     _mali_osk_timer_add(utilization_timer, _mali_osk_time_mstoticks(MALI_GPU_UTILIZATION_TIMEOUT));
101
102     mali_gpu_utilization_handler(utilization);
103 }
```

그러나 이는 커널 내부에서만 계산하고 활용하는 데이터일 뿐 mali parameter처럼 직접적으로 쉽게 확인하는 것은 불가능하다. 따라서 Kprobes라는 커널 메소드를 사용하여 리눅스 커널 모듈을 작성하였다. 모듈로 작성하였기 때문에 따로 커널 컴파일이 필요하지 않다는 장점이 있고, 또 Kprobes 중에서 jprobe를 사용하면 hooking하는 함수가 인자로 받는 parameter 값을 그대로 읽어 올 수 있으므로 용도에도 가장 적합하다. 작성한 test.ko라는 모듈은 mali_gpu_utilization_handler를 hooking 해서 그 parameter로 전달되는 Utilization 값을 읽어와 이를 커널 메시지로 출력한다. 다음은 해당 출력 부분만 따로 출력한 결과이다.

```
<4>[  610.805205] c0 [GPU] Utilization = 1
<4>[  611.805199] c0 [GPU] Utilization = 0
<4>[  618.685054] c0 [GPU] Utilization = 62
<6>[  618.710106] c0 [TSP] DVFS Off!
<4>[  619.685106] c0 [GPU] Utilization = 75
<4>[  787.420063] c0 [GPU] Utilization = 255
<4>[  788.420061] c0 [GPU] Utilization = 255
<4>[  789.420057] c0 [GPU] Utilization = 255
<4>[  790.420049] c0 [GPU] Utilization = 249
<4>[  791.420061] c0 [GPU] Utilization = 255
<4>[  792.420049] c0 [GPU] Utilization = 243
<4>[  793.420056] c0 [GPU] Utilization = 255
<4>[  794.420060] c0 [GPU] Utilization = 255
```

2.2.3 Experiment based on own application

이제 Utilization 기반의 모델링에 필수적으로 필요한 데이터들(단위 시간마다 계산되는 Utilization 값, 동작 클럭 및 전압 값)을 수집하는 방법을 마련하였으므로 실제로 유용한 데이터를 수집할 차례이다. 앞서 Mali-400MP의 동작 클럭 및 Utilization 값의 변화 등을 확인하기 위해 benchmark를 비롯해서 여러 가지 어플리케이션들로 테스트를 해보았지만, 이는 GPU에 맡기는 연산의 양을 직접 조절할 수 없고 이미 만들어진 대로만 사용해야 하기 때문에 유용한 정보들을 뽑아내는 데는 무리가 있다. 수집한 정보들로 전력 소비에 관한 공식을 유도해내기 위해서는 매우 적은 연산량부터 점차적으로 그 크기를 늘려가면서 테스트하는 과정이 필수적이기 때문이다. 따라서 본 연구를 위해 다음과 같은 어플리케이션을 새로 만들게 되었다.

실험 환경 설정

측정 장비로는 Monsoon Power Monitor를 사용하였다. Monsoon Power Monitor는 옆의 사진 속의 장비로, 대상 기기의 소비전력 측정이 가능하다. 또한 이와 함께 소프트웨어 조작을 통해 대상 기기에 원하는 만큼의 전압을 걸어줄 수 있어 실험 시 소비 전력을 측정하는데 핵심적인 장비라고 할 수 있다.

실험 대상 기기인 Galaxy S3는 AMOLED 액정을 가지고 있는 안드로이드 기반 스마트폰이다. AMOLED 액정을 가지고 있다는 것은 이번 실험과 밀접한 연관이 있는데, AMOLED를 비롯한 OLED 액정은 디스플레이에 나타내는 색에 따라 소비 전력이 각각 다르다는 특징이 있다. (이에 비해 일반 LCD 액정은 색 종류와 상관없이 밝기에 따라 소비 전력이 결정된다.) 따라서 GPUtest 어플리케이션을 동작시키고 Galaxy S3의 소비전력을 측정할 경우, 디스플레이에 나타나는 큐브 이미지의 색상에 따라 각 부분에서 서로 다른 소비전력이 발생하여 오차가 발생할 가능성이 높다.

또한, 디스플레이에서 나타내는 색상에 따른 소비전력 차이가 없다고 해도 기본적으로 디스플레이가 켜진 상태에서 실험을 진행할 경우 GPU만의 소비 전력을 구하기 위해서는 전체 소비 전력과 CPU가 소비하는 전력은 물론, 디스플레이가 소비하는 전력까지 따로 알아야만 한다.

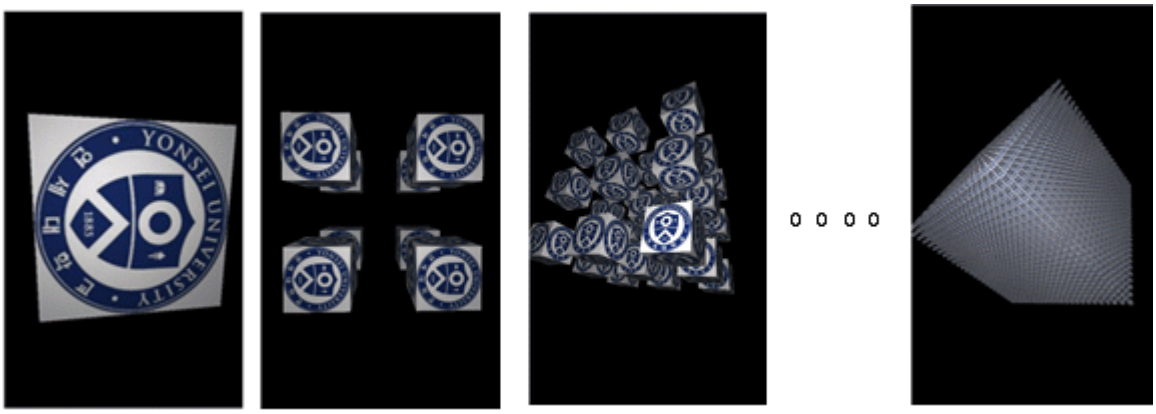
이 경우 'GPU가 소비하는 전력 = 전체 소비전력 – CPU가 소비하는 전력 – 디스플레이가 소비하는 전력' 이 되기 때문이다. 전체 소비전력은 위에서 언급한 Monsoon Power Monitor를 통해 직접 측정하는 것이지만, CPU가 소비하는 전력 및 디스플레이가 소비하는 전력은 모두 실제 측정값이 아닌 모델링을 통한 추정 값을 사용할 수밖에 없으므로, 모델링을 통한 추정 값이 비교적 정확하다고 해도 실험의 정확성 및 편리성을 위해서는 근사값의 수 자체를 줄이는 것이 더 효율적이다. 따라서 본 실험에서는 디스플레이를 끄고 진행하기로 하였다.

실험 어플리케이션

GPU의 소비전력 측정을 위해 기존에 있는 벤치마크 어플리케이션이나 기타 GPU를 사용하는 어플리케이션을 사용하지 않고 직접 어플리케이션을 제작한 이유는 보다 정확하고 체계적인 실험을 위해서는 어플리케이션을 수정하는 과정이 필요할 수밖에 없기 때문이다.

첫째로는, 보다 정확한 실험을 위해 디스플레이를 끄는 코드를 어플리케이션 소스 내에 추가하였다. 여기서 하나의 문제가 발생하였는데, 안드로이드 특성상 디스플레이가 꺼진 상태에서는 CPU와 GPU 모두 동작을 하지 않는다는 점이 문제였다. CPU의 경우는 다행히 어플리케이션 내에서 WAKELOCK 코드를 삽입하여 디스플레이가 꺼져도 동작할 수 있는 방법이 있었지만 GPU는 WAKELOCK을 사용하도 동작하지 않았다. 그래서 안드로이드 단이 아닌 커널 단에서 디스플레이를 끄고 WAKELOCK이 동작하도록 하는 모듈을 작성하였고, 이를 삽입하는 명령에 대한 코드를 어플리케이션에 추가하였다.

o o o



어플리케이션 이름은 GPUtest이다. GPUtest는 OpenGL ES 2.0을 기반으로 만들어졌는데, n^3의 큐브(정육면체)를 생성하여 특정 속도로 일정하게 회전시키는 작업을 수행한다. 처음 어플리케이션을 실행하면 간단한 타이틀 화면이 나오는데, 여기에 전체 큐브 수를 결정할 한 변 당 큐브 수, 회전 주기(ms)를 입력할 수 있다. 여기서 설정한 값들에 따라 큐브의 수 및 큐브의 회전 속도가 바뀌므로 원하는 데이터를 뽑아내기 위한 실험을 편리하고 반복적으로 수행할 수 있다.

실험 조건

전체적인 실험은 큐브들의 회전 주기를 100ms로 고정시키고, 한 변 당 큐브의 수를 1개부터 13개까지 점차 증가시키는 방향으로 진행하였고, Mali-400MP GPU의 네 가지 DVFS(Dynamic Voltage and Frequency Scaling) step(:0~3)에 대하여 각각 따로 실험하였다.

각각의 케이스별로 5번 실험하고 그 평균값을 데이터로 취하였으며, CPU는 1개의 코어만 동작시키고 Full DVFS Bus 기준으로 실험을 진행하였다.

실험 결과

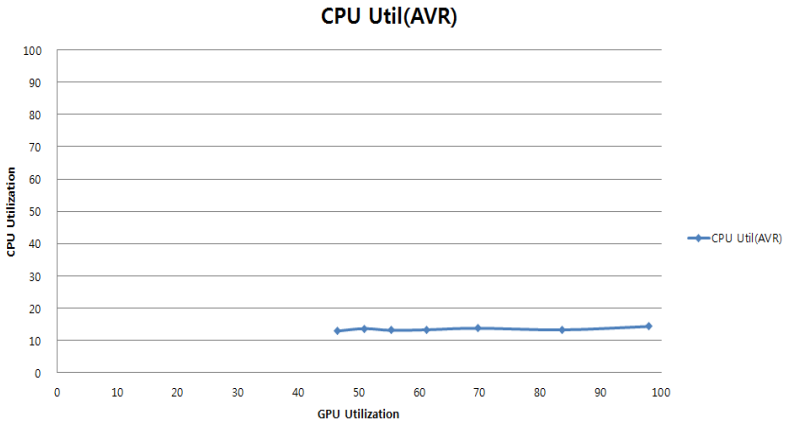
래의 표는 특정 DVFS step(frequency)에 대하여 한 변 당 큐브 수를 1개부터 13개까지 증가시켰을 때 GPU의 Utilization 값을 측정한 결과를 정리한 것이다. (회전 주기 : 100ms)

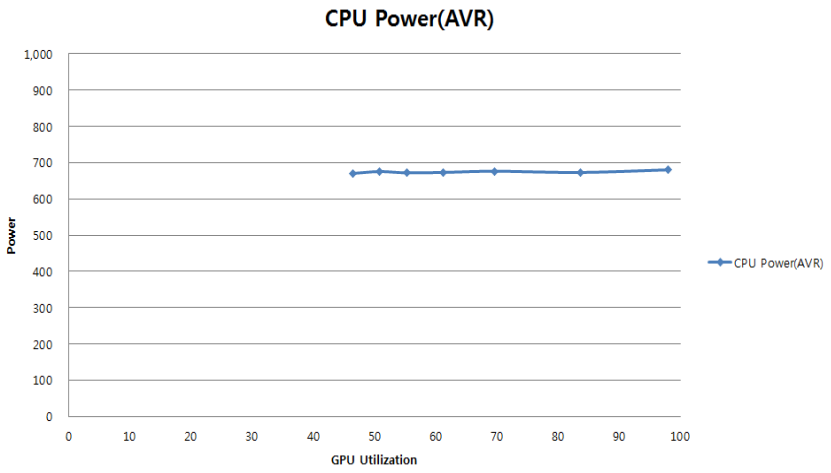
Cube Count	Step0 Value	Step1 Value	Step2 Value	Step3 Value
1	130(130-133)	107(106-109)	100	92(91-93)
3	141(138-142)	114(112-114)	Not Changed	Not Changed
4	150(149-151)	120(119-121)	Not Changed	Not Changed
5	166(164-168)	128(126-129)	115(113-117)	103(102-105)
6	186(183-188)	140(137-141)	125(122-128)	110(109-112)
7	205(205-209)	156(155-158)	137(135-139)	119(119-121)
8	Dropped	171(169-173)	151(150-151)	131(130-132)
9	Dropped	181(179-182)	151(148-151)	Dropped
10	Dropped	202(202-204)	170(168-171)	141(141-143)
11	224(224-227)	227(227-228)	193(191-195)	160(159-160)
12	Not Changed	234(232-238)	215(211-215)	182(179-183)
13	Not Changed	Not Changed	220(200-228)	206(206-208)

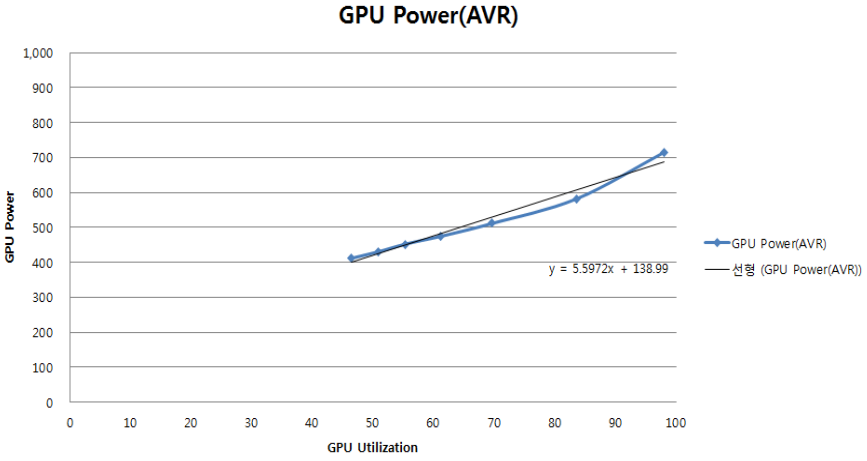
이 표를 살펴보면, 일부 예외를 제외하고는 큐브의 수를 증가시킬수록 각 step별로 GPU의 Utilization 값이 점차 증가하는 경향이 있다는 것을 알 수 있다.

다음은 각각의 실험 케이스들의 GPU Utilization(100 기준 환산값) 변화에 따라 CPU Utilization과 CPU 소비전력, 그리고 GPU 소비전력이 어떻게 변화하는지를 그래프로 나타낸 결과들이다.

Step 0(160Mhz)

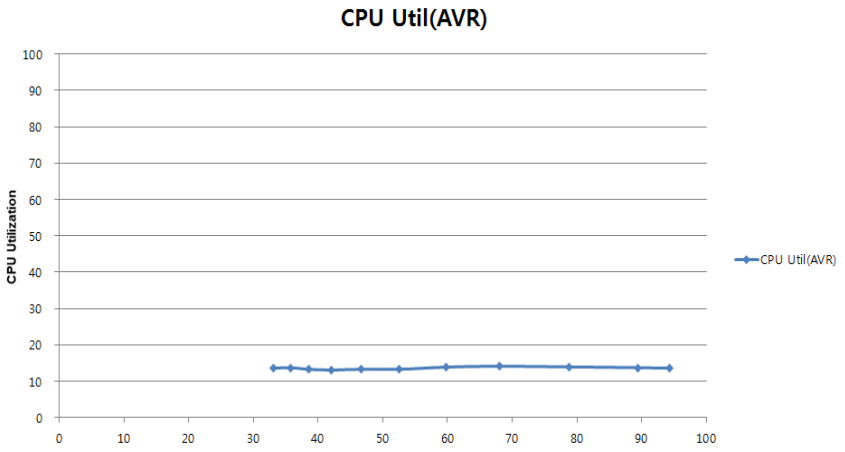


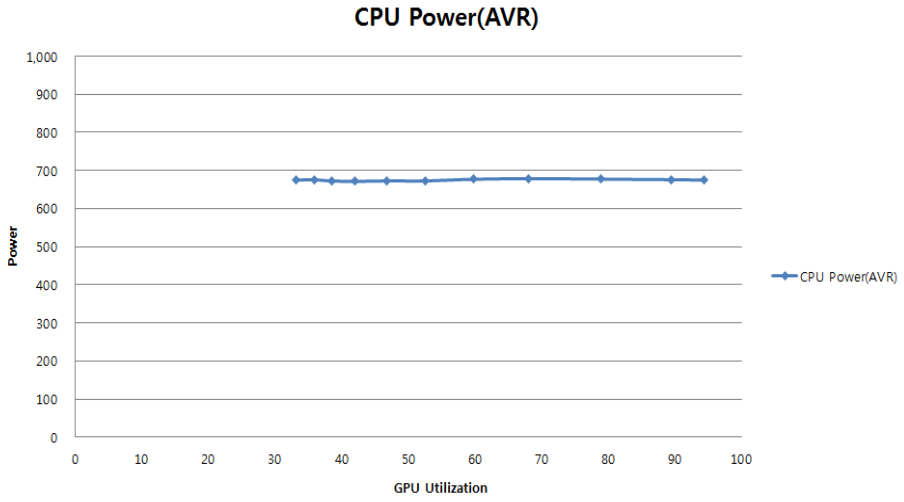


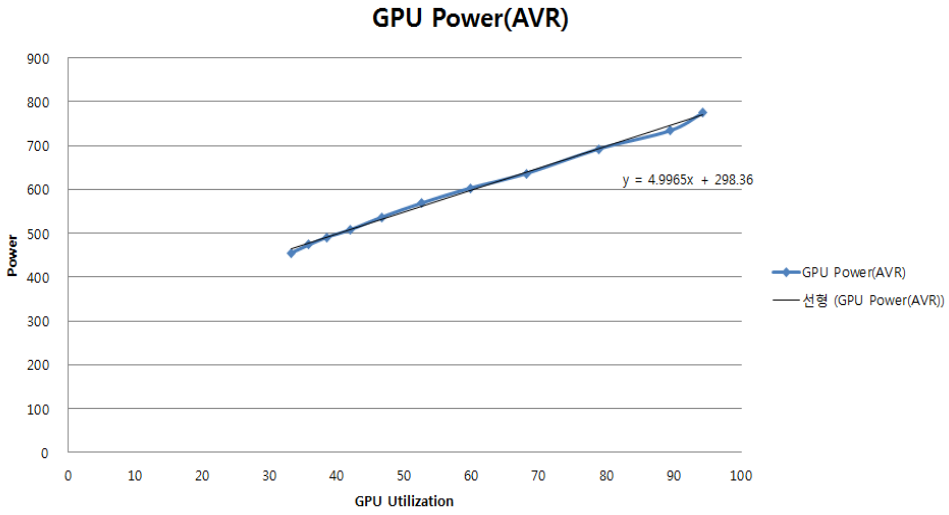


ㅇㅇㅇ

Step 1(260Mhz)

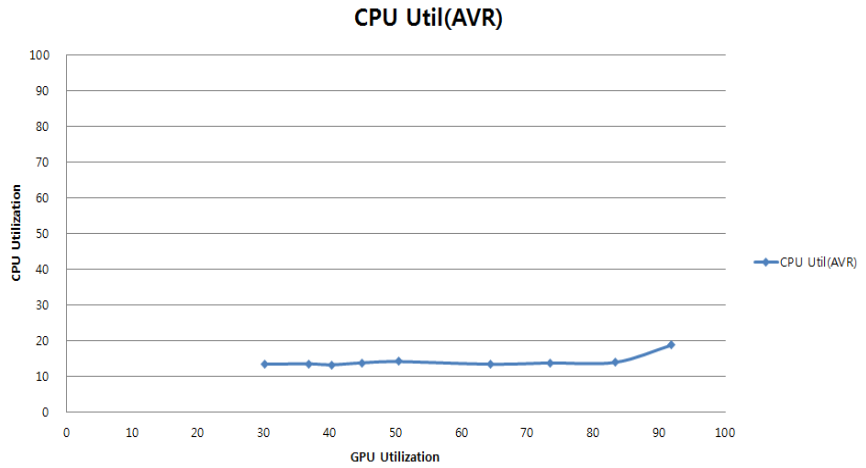


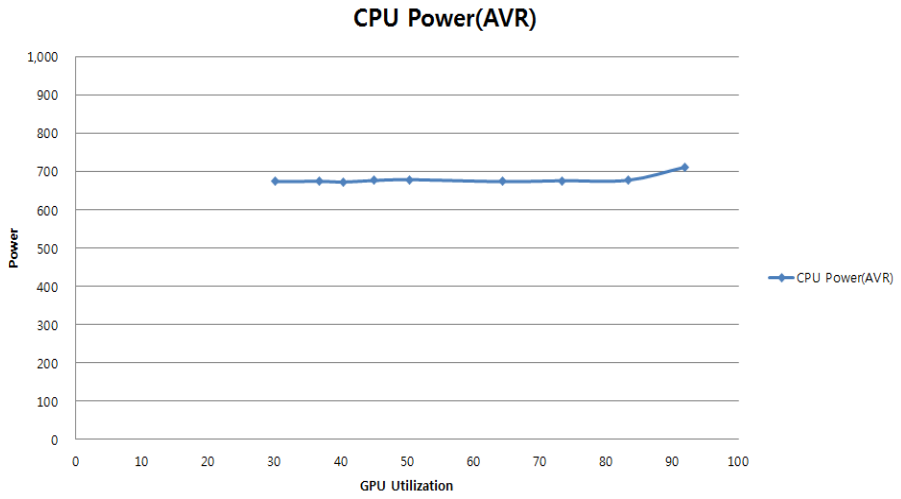


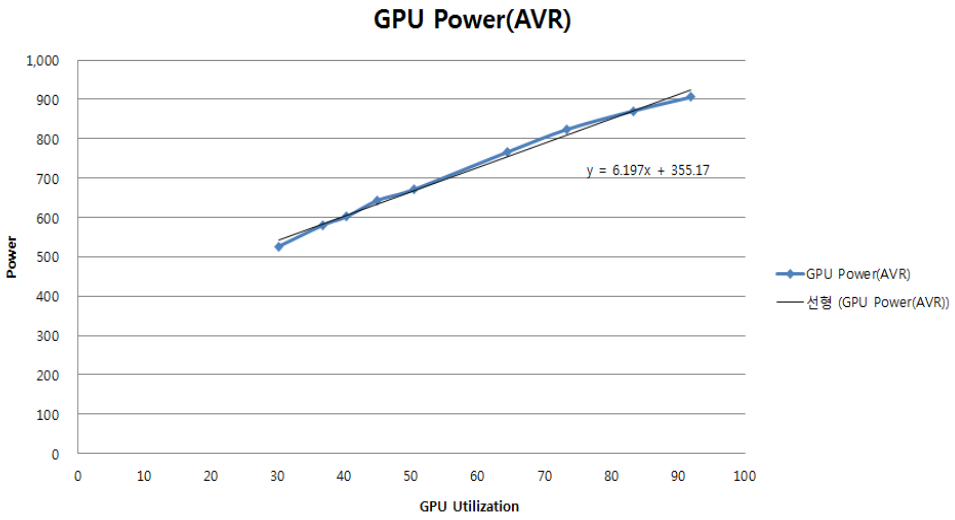


ㅇㅇㅇ

Step 2(350Mhz)

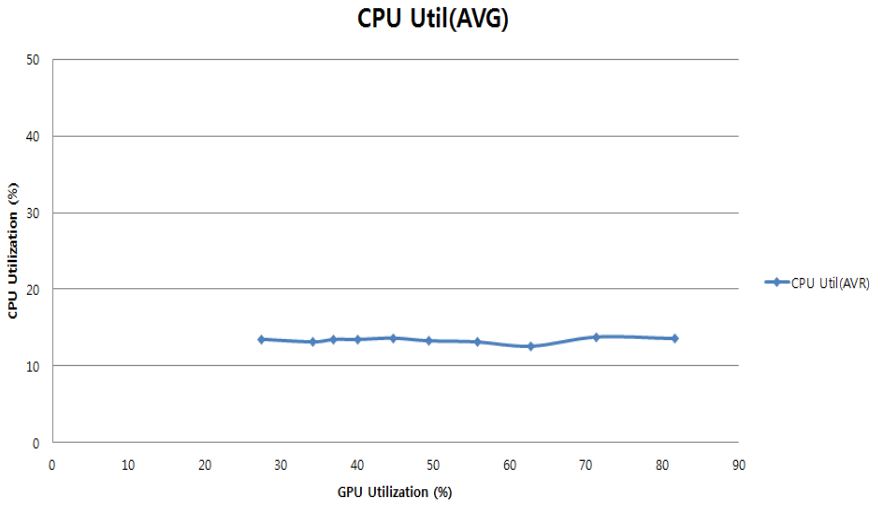


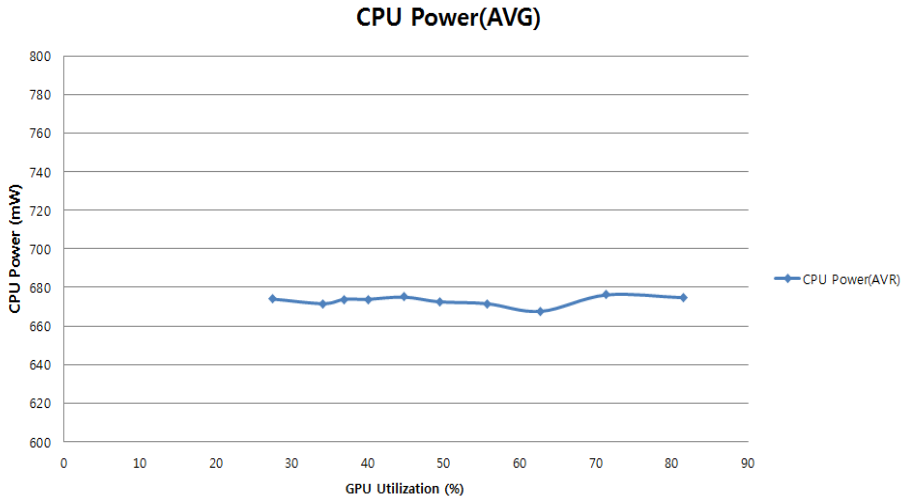


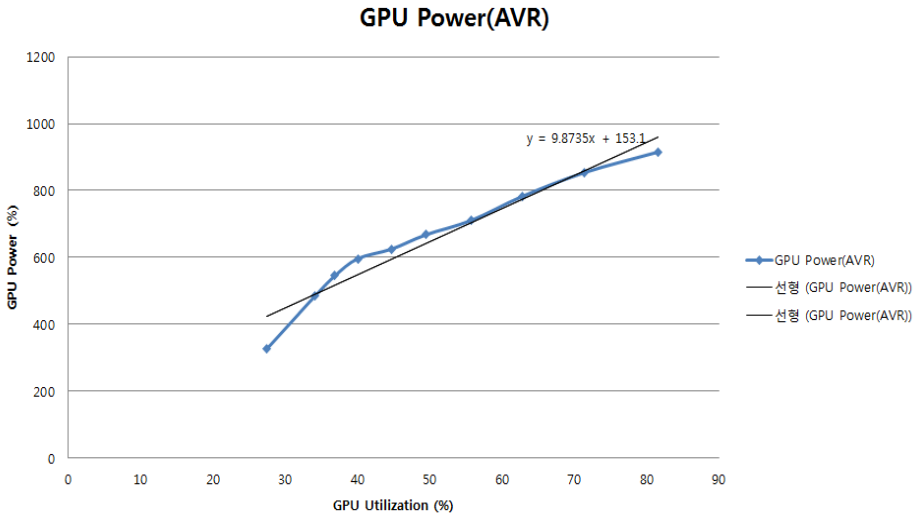


ㅇㅇㅇ

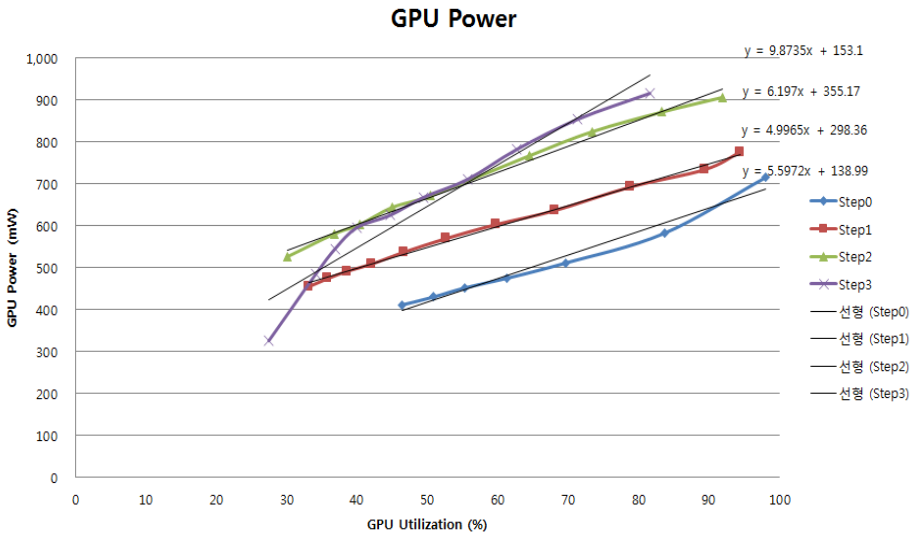
Step 3(440Mhz)







Overall



실험 결과 분석

위 실험 결과는 각 step별로 GPU Utilization vs CPU Utilization, GPU Utilization vs CPU Power, GPU Utilization vs GPU Power에 대한 그래프를 나타낸 것이다.

이 세 종류의 그래프들은 step(frequency)에 따라 데이터 값의 차이는 있지만, 그 경향은 step과 상관없이 나타난 것을 확인할 수 있다.

먼저 GPU Utilization vs CPU Utilization 그래프들을 살펴보면, GPU Utilization이 점차 증가함에도 불구하고 CPU Utilization은 비교적 일정한 것을 확인할 수 있다. 이는 GPUtest 어플리케이션의 작업이 대부분 GPU에 의해서 처리되기 때문에 작업량이 증가해도 CPU가 처리하는 양은 크게 변하지 않기 때문이다.

GPU Utilization vs CPU Power도 마찬가지로 GPU Utilization이 점차 증가함에도 불구하고 CPU Power가 비교적 일정한 것을 확인할 수 있는데, 이는 Utilization 기반의 모델링에서는 Utilization에 따라 Power가 결정되기 때문이다. CPU Utilization이 비교적 일정하게 유지되었으므로 당연히 CPU Power도 일정하게 유지된다.

(GPU 관련 실험에서는 1개의 CPU core만 동작시키므로, 기존의 Utilization 기반의 CPU 모델링을 사용하였다)

마지막으로 GPU Utilization vs GPU Power의 그래프는 GPU Utilization이 증가함에 따라 GPU Power도 동일하게 증가하는 linear(선형)한 결과가 나타남을 확인할 수 있는데, 이는 GPU Utilization과 GPU Power의 관계를 일차함수로 표현할 수 있다는 것을 의미하고 그 일차함수 식이 바로 우리가 구하려는 GPU Power에 대한 모델링 공식이다.

ㅇㅇㅇ

모델링 공식(최종 모델링 공식)

: 데이터 정리를 통하여 얻은 GPU 소비전력에 대한 모델링 공식은 다음과 같다.

$$P_{GPU} = \sum_{i\_step0}^{step3}(\beta_i * GPU\_Utilization + Base\ Power_i)$$

여기서 β는 각 step별로 정해지는 상수이며, GPU Utilization은 퍼센트(%, 100 기준으로 환산) 값이고, Base Power 역시 각 step별로 정해지는 기본 전력값이다.

각 step별로 실제 β와 Base Power를 대입한 모델링 공식은 아래와 같다.

Step 0 : 5.5972 * GPU Utilization + 138.99

Step 1 : 4.9965 * GPU Utilization + 298.36

Step 2 : 6.197 * GPU Utilization + 355.17

Step 3 : 9.8735 * GPU Utilization + 153.1

3. 진행 상황

3.1 Multi-Core CPU

연구실에서 진행하는 모델링의 경우는 앞으로 더 실험을 진행해서 모델링을 확고히 할 예정

우리는 현재까지 진행한 연구 데이터 내에서 연구실과는 별도로 모델링을 정하기로 했고 현재 Cache 부분의 모델링에 대해서 논의 중이고 모델링이 완료되면 상용되는 어플리케이션을 통해 runtime 측정을 진행할 예정

어플리케이션은 되도록 multithreading 기술이 반영되어있는 것을 찾아서 여러 개의 코어가 동시에 동작하고 있을 때 측정이 제대로 이루어지는지를 확인해 볼 것이다.

최종 발표 전에 모델링을 완성하고 해당 모델링을 뒷받침하는 데이터를 정리할 계획

3.2 GPU

다른 어플리케이션으로 모델링 검증하기 (필요할 경우 별도의 어플리케이션 새로 작성)

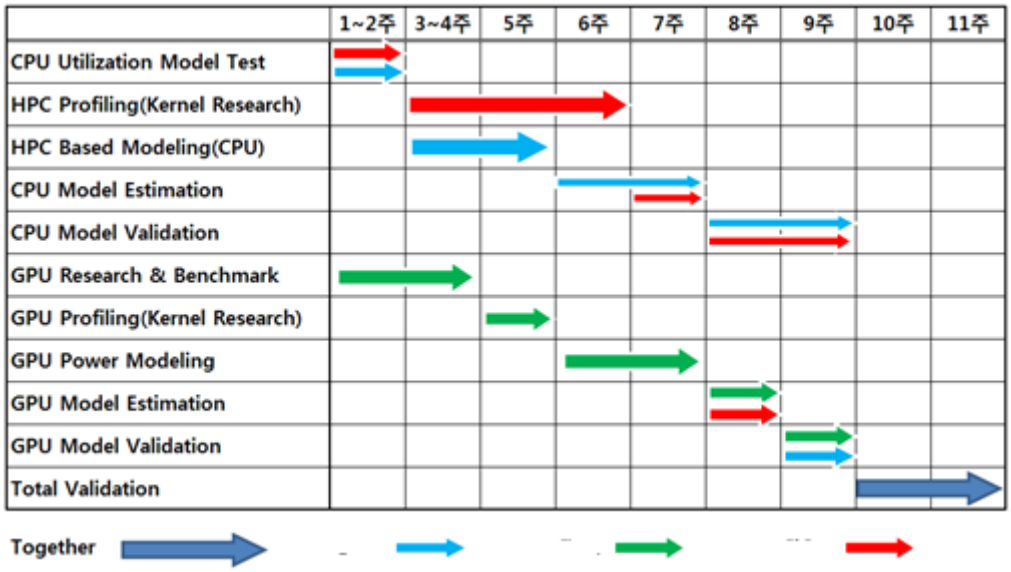
현재의 모델링은, OpenGL ES 2.0을 기반으로 하여 여러 개의 큐브를 생성하고 이를 회전시키는 작업을 수행하는 테스트 어플리케이션을 통해 세워진 것이다. 결과 그래프에서도 확인 할 수 있듯이 우리는 비교적 성공적인 형태의 리니어한(선형화된) 모델링을 얻을 수 있었지만 다른 그래픽 처리를 할 경우에도 우리가 세운 모델링 공식이 유효한지 검증하는 과정이 필요하다.

따라서 다음 실험에서는 GPUtest와는 다른 그래픽 처리를 하는 테스트 어플리케이션을 사용할 예정이며, 현재 적당한 테스트용 어플리케이션을 물색 중이다.

테스트용 어플리케이션은 되도록 오픈 소스 중에서 찾고 있는데, 그 이유는 앞서 '실험 환경 설

정’ 부분에서 설명한 대로 보다 정확한 실험을 위해서는 디바이스의 화면을 끄고 GPU 및 CPU 가 동작하도록 하는 모듈과 실험의 자동화를 위한 모듈을 삽입하는 코드를 어플리케이션 코드 내에 삽입해야하기 때문이다.

4. 일정 및 역할 배분



현재 10주차까지 진행된 상태이다.