듀얼 페이즈 구조의 멀티 코어 GP-GPU를 이용한 픽셀 셰이딩

김준서* · 박태룡* · 이광엽*
*서경대학교 컴퓨터공학과

The Pixel Shading on Multi Core GP-GPU with Dual Phase Architecture

Jun-seo Kim* · Kwang-yeob Lee*
*Seo Kyeong University, Dept. Computer Engineering
E-mail : Junseo@skuniv.ac.kr

요 약

최근 프로세서가 클럭 향상의 한계에 부딪힘에 따라, 프로세서의 성능을 향상시키기 위해 멀티 코어 기반의 병렬처리를 이용한 방법들이 제안 되고 있다. 본 논문은 여러개의 연산기를 한 명령어 사이클에 동시에 사용할 수 있는 MIMD(Multiple Instruction, Multiple Data) 구조를 가지며, Scratch Counter를 이용해 멀티 코어와 멀티 스레드의 작업을 할당하는 구조의 GP-GPU(General Purpose - Graphics Processing Unit)를 활용해 멀티 코어, 멀티 스레드 환경에서의 효율적인 픽셀 셰이딩 방법을 설계 하였다. 선형 안개 픽셀 셰이딩의 경우 싱글코어에서 18.3 FPS이며 4개의 멀티코어 GP-GPU에서는 4배가 증가한 73.2 FPS 결과를 얻었다.

키워드

듀얼 페이즈, 멀티 코어, GP-GPU, 픽셀 셰이딩

Ⅰ. 서 론

최근 수년간의 컴퓨터 산업은 중요한 과도기를 맞이하게 되었다. 그간의 컴퓨터 산업의 성장에도 불구하고 프로세서의 성능은 한계에 부딪혔다. 기존의 공정 미세화를 통한 동작속도 향상이 한계점에 도달하였고, 이에 따라 프로세서의 동작 주파수는 수년째 3GHz대를 벗어나지 못하고 있다.[1] 기존의 공정 미세화를 통한 성능 향상이 어려워지자 사용자들의 요구 성능을 충족시키기 위해 멀티 코어 기술이 사용되었다.

멀티 코어 기술의 등장으로 프로세서는 무어의 법칙을 만족하면서 프로세서의 동작 성능을 크게 향상시킬 수 있었지만, 이론적인 연산성능의 향상에도 불구하고 실 사용자가 체감하는 성능 향상 폭은 크지 않았다. 기존의 소프트웨어는 병렬처리를 고려하지 않은 방법으로 개발되었고, 기존에 단일 코어, 단일 스레드 기반의 환경에서 개발된 이러한 소프트웨어는 멀티 코어 프로세서의 자원을 모두 활용하지 못하였기 때문이다. 멀티 코어 프로세서가 제 성능을 내기 위해서는 소프트웨어 역시 멀티 코어 구조를 고려하여 병렬처리가 가능하게 설계 되어야 한다.

하지만 기존의 단일 스레드 프로그램의 작성에 익숙한 개발자에게 병렬 처리가 고려된 멀티 스레드, 멀티 코어 프로그램은 난해한 작성 방법으로 인해 접근이 쉽지 않았고,[2] 특히 응용 범위가 범용 프로세서보다 적으면서도 병렬처리가 가장 많이 사용되어지는 GPU 프로그램 작성에서는 이러한 현상은 더욱 심화되어 병렬 처리 프로그램 작성에 많은 어려움이 있었다.

그러나 이러한 어려움에도 불구하고 멀티 코어 프로세서 기술이 발전하고 보편화됨에 따라 이제는 PC뿐만 아니라 휴대폰, PMP와 같은 모바일 기기에서도 멀티 코어를 도입하려는 노력이 있으며[3] 최근에는 멀티 코어뿐만 아니라 높아진 소비자의 요구사항을 충족시키기 위해 모바일 GPU까지 사용되어진다.

본 논문은 멀티 스레드, 멀티 코어 GP-GPU를 활용하여 픽셀 셰이딩 연산을 병렬 처리하여 성

Acknowlegement
본 논문은 지식경제부 출연금으로 ETRI 시스템반도체진흥센터에서 수행한 시스템반도체 융복합형 설계 인재양성사업의 연구결과입니다.

능을 높이는 방법을 연구한다.

II. 멀티 코어 GPU의 구조

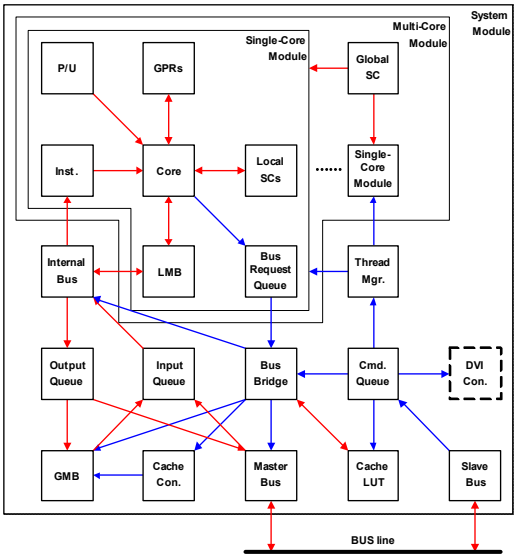


그림 1. 시스템 구성도

그림 1은 멀티 코어 시스템 구성도이다. 각 싱글 코어 모듈에는 스레드 컨트롤을 위한 Local SCs(Local Scratch Counter)와 데이터 전송을 위한 Bus Request Queue, GPRs(General Purpose Resisters), Instruction Registers가 존재 한다. 각 싱글 코어는 다시 멀티 코어 컨트롤을 위한 Global SCs(Global Scratch Counter)에 연결되어 있다.

각 스레드는 Local SCs로, 각 코어는 Global SCs로 각 코어와 스레드 스스로가 자신의 작업량을 할당하는 구조이다.

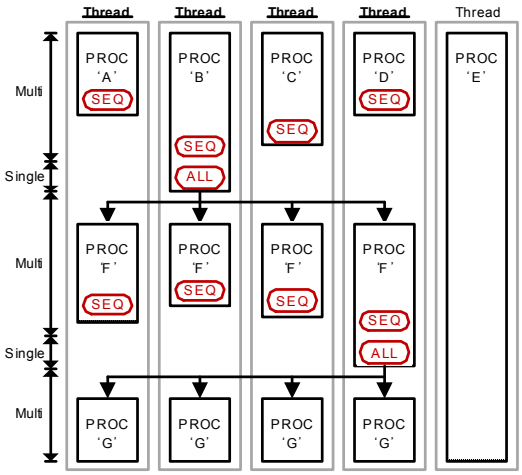


그림 2. 멀티스레드 자기 관리 방법

그림 2는 GPU의 멀티스레드를 효율적으로 자기 관리 하기 위한 구조를 나타낸다. 이 구조는 별도의 제어 유닛이 필요 없고 크리티컬 섹션이 필요 없으며 단순한 구조로 작업을 빠르게 트랜잭션 할 수 있는 구조로 되어있다.

또한 여러개의 연산기를 병렬로 두는 듀얼 페이즈 명령어 구조를 사용하여[4] 최대 4개의 소스 오퍼란드(Source operand)와 최대 2개의 데스티네이션 오퍼란드(Destination operand)를 가지며, 이러한 구조의 장점으로는 32비트 길이의 소스 오퍼란드를 하나의 실행 사이클에 최대 16개까지 연산할 수 있으며 데스티네이션은 32비트 길이의 소스 오퍼란드를 최대 8개까지 저장할 수 있는 구조이다. 또한 각각의 페이즈 마다 서로 다른 연산기를 사용할 수 있는 구조이다.

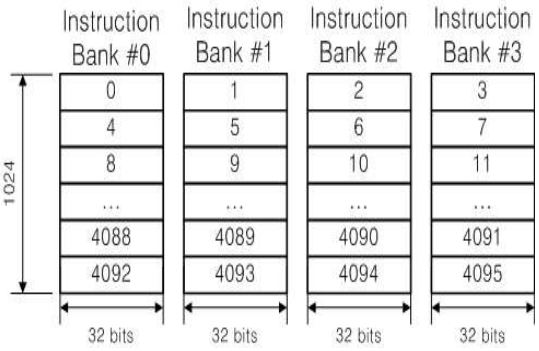


그림 3. 명령어 레지스터 구조

그림 3은 명령어 레지스터 구조로 최대 128비트 가변길이 명령어의 추출(Instruction Fetch)을 위한 뱅크 구조이다. 서로 다른 네 개의 뱅크에 저장되어 있는 유닛 명령어들의 집합으로 각 명령어의 End Bit에 따라 최대 128비트까지 동시에 명령어 추출이 가능하다. 또한 동시에 추출된 명령어로 서로 다른 연산기를 동시에 사용하는 병렬 처리가 가능하다.

이러한 병렬 처리가 가능한 MIMD 구조는 일반적인 SIMD(Single Instruction Multiple Data) 구조에 비해 명령어 실행 사이클을 대폭적으로 줄일 수 있는 구조이다.

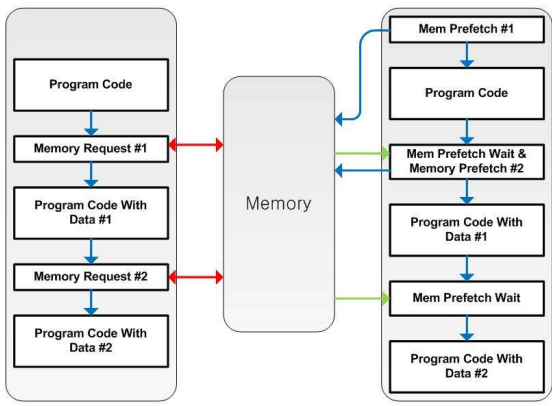


그림 4 메모리 프리펫치 기능

그림 4는 병렬처리의 효율을 높이기 위한 메모리 프리펫치 기능을 도식화 한 것이다. 메모리 전송시의 대기시간을 줄이기 위하여 메모리 프리펫치 기능을 사용하였다. 메모리 프리펫치 기능을 사용함으로써 메모리 접근시간동안 프로세서가 대기상태로 들어가지 않고 데이터가 필요한 이전 시점에 데이터를 요청하여 메모리 전송 대기 시간에 프로세서가 다른 작업을 하게 함으로써 병렬 처리의 효율을 높였다.

III.픽셀 셰이딩

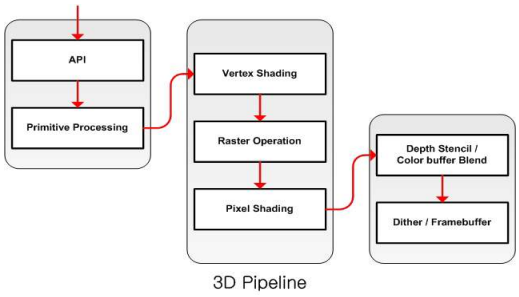


그림 5. OpenGL ES 2.0의 3D 파이프라인

그림 5에서 보여지는 것과 같이 전체 3D 그래픽 파이프라인에서 픽셀 셰이더가 하는 일은 Rasterizer 단계 이후의 화소 연산이다. 픽셀 셰이더는 래스터라이저 이후의 픽셀에 대해 각종 3D 효과를 연산 한다.

이 픽셀 셰이딩 과정은 모든 화소에 대해 연산하기 때문에 연산량이 많고 또한 화소를 메모리를 통해 전송 받기 때문에 메모리 전송량 또한 부담이 되는 과정이다. 그러나 픽셀 셰이딩 과정에서 각 픽셀은 서로에 대한 데이터 의존성이 없으며 이로 인해서 각 픽셀은 독립적으로 연산이 이루어지기 때문에 병렬 처리에 유리하며 병렬처리시에 동작 성능의 향상을 기대할 수 있다.

본 논문에서는 픽셀 셰이딩을 각각 멀티 코어, 멀티 스레드 환경과 싱글코어, 싱글 스레드 환경에서 구현 해 보았으며 메모리 전송을 줄이기 위해 스레드마다 한번에 16개의 화소와 그에 따른 Varying Value를 전송하였다.

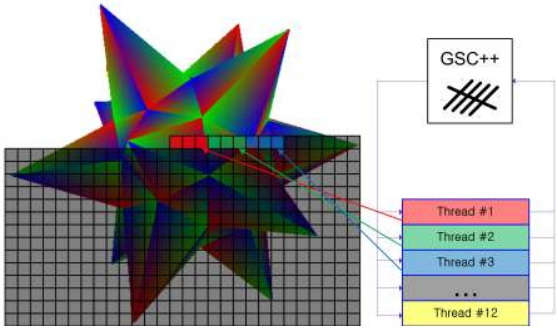


그림 6. 스레드별 픽셀 셰이딩 작업 할당

그림 6은 픽셀 셰이딩의 작업 과정과 각 스레드의 작업 할당을 도식화 한 것이다. 각 스레드의 작업 할당은 글로벌 스크래치 카운터를 사용해 작업량을 할당 하며, 한 스레드가 16개의 픽셀과 Varying Value를 가져와 이를 MIMD구조의 듀얼 페이즈 명령어 구조를 이용해 명령어 실행 사이클을 줄일 수 있었다.

IV. 검증

GPU는 RTL 수준에서 100MHz로 동작 하게 설정되었고 성능 측정은 RTL 수준으로 시뮬레이션이 행해진 버텍스 셰이딩과 픽셀 셰이딩 중 픽셀 셰이딩에 한하였다. GPU는 4개의 멀티 코어와 코어마다 각각 12개의 스레드로 이루어진 GPU 환경에서 시뮬레이션이 수행되었다.

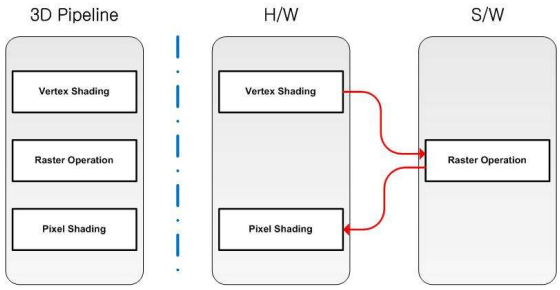
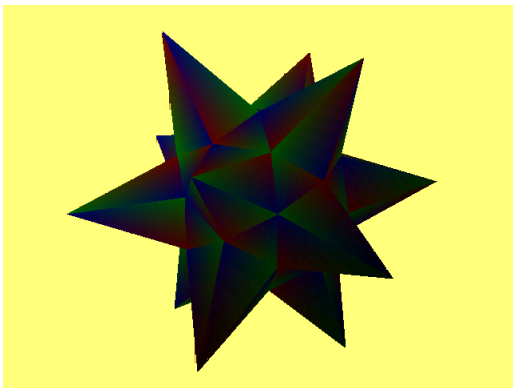


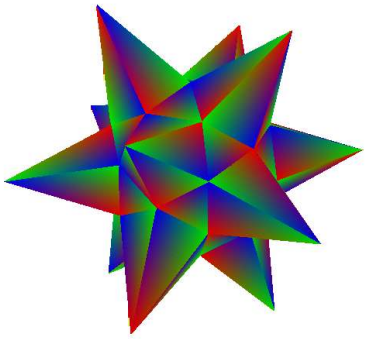
그림 7. 시뮬레이션 환경

그림 7과 같이 버텍스 셰이딩과 픽셀 셰이딩은 GPU 명령어로 작성해 RTL 수준으로 시뮬레이션 하였으며, 래스터라이저는 System Verilog의 DPI(Direct Programming Interface)를[5] 활용해 C 언어로 작성하였다.

시뮬레이션은 Mentor Graphics 사의 Model SIM 6.0c를 이용하였다.



(a) MaxDistance 1.4, Min Distance 0.2



(b) MaxDistance 3, Min Distance 0.2

그림 8. 선형 안개(Linear Fog) 예제

그림 8은 픽셀 세이딩 예제로 사용된 선형 안개 연산의 결과물이다. 선형 안개의 세이더 코드는 다음과 같다.

표1. 선형 안개 코드

Linear Fog Shader Code[6]

```
float computeLinearFogFactor()
{
 float factor;
 factor = (u_fogMaxDist = v_eyeDist) /
      (u_fogMaxDist - ufogMinDist);
 factor = clamp(factor, 0.0, 1.0);
 return factor
}
void main(void)
{
 float fogFactor = computeLinearFogFactor();
 vec4 fogColor = fogFactor * u_fogColor;
 vec4 baseColor = texture2D)baseMap,
v_texcoord);
 gl_FragColor = baseColor * fogFactor +
        fogColor * (1.0 - fogFactor);
}
```

표 2. 픽셀 세이딩 결과 성능치

	소요 시간	FPS
Multi Core	13m 3u 653ns	73.2
Single Core	54m 537u 520ns	18.3

먼저, 멀티 코어의 경우 640 * 480의 해상도에 대해 그림 8의 선형 안개 연산을 했을 때 표 2에서 보이는 것과 같이 하나의 프레임에 14m 3u 150ns가 소요 되었고 이는 FPS(Frames per Second)로 환산시 71.4 프레임의 성능을 보였다. 여기에 메모리 프리펫치 기능을 사용했을 때는 13m 653u 520ns가 소요되어 FPS 환산 시 73.2 프레임의 성능을 보였다. 또한 싱글코어 멀티 스레드의 경우 54m 537u 520ns가 소요 되었고, FPS 환산시 18.3프레임으로 환산되어 픽셀 세이딩 과정을 멀티 코어, 멀티스레드 환경을 고려하여 병렬처리 하면 약 4배의 성능 향상이 있었음을 알 수 있다.

Ⅴ. 결론

3D 그래픽스 파이프라인 중 픽셀 세이딩 연산은 많은 양의 화소 연산과 그에 따른 메모리 전송으로 인해 부담이 되지만, 각 픽셀간의 데이터 의존성이 적고 이에 따라 각 화소의 연산이 독립적으로 이루어질 수 있으며 메모리 접근을 줄이기 위해 한번에 16개의 화소와 Varying Value를 버스트 모드로 전송하였다.

본 논문은 3D 그래픽 파이프라인 중 멀티 코어, 멀티 스레드 환경 하에서 병렬처리를 고려하여 픽셀 세이딩을 구현하였다. 해상도의 화면에 대해 선형 안개 연산 시 멀티 코어 환경에서는 73.2 프레임으로, 싱글코어 환경에서는 18.3 프레임으로 동작 하는 것을 확인할 수 있었으며, 데이터 의존성이 적고 연산량이 많은 픽셀 세이딩 과정을 병렬처리 하면 성능 향상을 기대할 수 있고, 실제로 선형 안개 과정을 병렬 처리 시 약 4배의 성능 향상이 있었다.

참고문헌

[1] 정형기, 능동형 스레드 관리 기법을 적용한 멀티 스레드 멀티 코어 GP-GPU 설계

[2] Tim Sweeney, Keynote, High Performance Graphics Conference 2009

[3] ARM, Cortex A15 MP Core, http://www.ARM.com

[4] Woo-Young Kim, A Design of a Shader based on the Variable-Length Instruction for a Mobile GP-GPU

[5] http://www.systemverilog.com/

[6]Aaftab Munshi, OpenGL ES2.0 Programming Guide