가변길이 SIMD구조 쉐이더 명령어 및 컴파일러 설계

곽재창* · 박태룡**

Design of Compiler & Variable-Length Instructions for SIMD Structured Shader

Jae-Chang Kwak* · Tae-Ryoung Park**

요 약

본 논문에서는 3차원 그래픽 쉐이더 3.0 API를 지원하는 쉐이더 명령어 및 컴파일러를 설계하고 그 결과를 평가한다. 기존의 명령어와는 달리 가변길이의 명령어 구조를 제안하고 명령어의 길이를 줄여 SIMD(Single Instruction Multiple Data)구조의 그래픽 프로세서의 하드웨어 크기를 줄일 수 있다. 가변길이 및 2 페이즈 구조의 명령어를 지원하며 ESSL(ES Shading Language) 수준에서 쉐이더 프로그램이 가능한 쉐이더 컴파일러의 설계를 수행하였다. 명령어와 컴파일러 설계 결과를 검증하기 위하여 크로노스그룹에서 제안하는 Conformance Test를 수행하였다. 그 결과로 제공하는 기본 GL 쉐이더의 기능 16개를 비교하여 보았을 때 전체 평균 37%가 줄어드는 것을 알 수 있다.

ABSTRACT

Shader instructions and Compiler are designed for supporting 3D graphic shader 3.0 API. Variable-length instructions are proposed to reduce the size of hardware of graphic processor in SIMD structure by shortening the length of instructions. The designed shader compiler supports variable and two phased structured instructions, and can be programmable at ESSL level. Conformance Test proposed by Khronos group is accomplished to verify the design result of instructions and complier. The test result shows overall average 37% performance improvement at the 16 functions of basic GL shader.

키워드

3차원 그래픽, 쉐이더, 쉐이더 명령어, 쉐이더 컴파일러

Key word

3D graphics, Shader, Shader Instruction, Shader Compiler

* 서경대학교 컴퓨터과학과
** 서경대학교 컴퓨터공학과 (교신저자, trpark@skuniv.ac.kr)

접수일자 : 2010. 06. 15
심사완료일자 : 2010. 07. 15

Ⅰ. 서 론

최근 3차원 그래픽 프로세서는 쉐이딩(shading)의 효과를 높이기 위하여 명령어 기반의 프로그램 방식이 도입되고 있다. 크로노스그룹의 OpenGL 2.0 API가 대표적인 예이다[1]. 이와 같은 명령어 기반의 쉐이딩 프로세서는 상위수준의 API를 통한 사용자 프로그램을 지원하기 위하여 쉐이더 컴파일러를 필요로 하며 컴파일러에서 생성된 명령어는 크로노스그룹의 테스트를 만족해야만 완성이 될 수 있다.

본 논문에서는 OpenGL ES 2.0 API를 지원하는 명령어를 새로운 구조로 설계하고 사용자 프로그램이 가능한 컴파일러를 제작하였다. 또한,설계된 명령어와 컴파일러의 타당성을 입증하기 위하여 크로노스그룹의 conformance 테스트를 수행하여 그 결과의 우수성을 입증하였다.

기존에 설계하였던 쉐이더(Shader) 프로세서는 64비트 RISC 명령어 타입의 SIMD 구조를 가지고 있었으며, Shader 1.0 모델에 특화되어 설계하였다. Dot-product에 최적화된 연산기의 3 stage 파이프라인 구조를 가지고 있으며, 모든 명령어가 1 clock 마다 처리될 수 있도록 하였다. 명령어 형식은 MO-LUT(Micro-Operation Look-Up Table)를 이용하여 명령어 셋을 소프트웨어에서 변경하거나 추가할 수 있는 구조를 채택하고 있었다[2].

기존의 설계에는 연산기에 특화 되어 있어 구조가 단순하였으나 연산 명령 이외에 소스가 4D vector 방식의 3개 오퍼랜드를 사용하고 다양한 레지스터 그룹이 있어 이를 컨트롤하기 위해 64비트 명령어 구조를 채택하고 있었으나, Shader 2.0 이상에서는 이보다 더 다양해진 레지스터 그룹들과 Dynamic Flow Control 지원을 위한 새로운 명령어들이 추가되면서 명령어 필드의 추가적인 길이 확장과 하드웨어의 복잡성 증가가필수적인 사양이 되었다.

하드웨어 레벨에서 이를 모두 지원하기 위해 많은 자원이 필요하게 되었으며, 이로 인해 예상되는 단점은 하드웨어의 크기 증가, 명령어 필드 낭비와 명령어 필드 확장성에 대한 향후 호환성 결여 등의 문제점이 예상되며, 기존의 Shader 버전에서의 MO- LUT의 활용성의 결여, SIMD 구조의 기본적인 문제인 많은 연산기들을 구비한데 비하여 효율성이 부족한 점이 지적되었다.

이것을 해결하기 위해 보다 높은 성능과 호환성 그리고 빠른 설계를 위해 소프트웨어와 하드웨어 디자이너가 레지스터 관리와 명령어의 복잡성을 나누어 처리하는 시스템 레벨 디자인(System Level Design)이 필요하게 된다. 따라서 본 논문에서는 이전의 연구와 비슷한 자원으로 Shader 3.0을 지원하고 보다 빠른 성능과 높은 호환성과 연산기의 효율성을 증가시킨 프로세서를 개발하였다.

Ⅱ. 제안하는 쉐이더 명령어

2.1 기존 명령어 구조

기존에 개발한 버텍스 쉐이더(Vertex Shader) 명령어 구조는 그림 1과 같이 필드가 고정된 SIMD 64bit RISC 타입의 명령어 구조였다. 이러한 방식의 장점은 디코딩하는 필드가 고정되어 있어 단순하고 설계가 쉽다는 장점이 있으나 명령어 표현상에 제약이 많고 향후 명령어 추가에 대해여 취약하다는 단점이 있다[3].

기존의 구조에서는 그림 2와 같이 단순한 레지스터 그룹으로 이루어져 있었다. Shader 3.0 모델에서는 이에 Dynamic-branch, looping 과 같은 기능이 추가되면서 그림 3과 같이 코어에 많은 레지스터 그룹들이 추가되었다. 이러한 구조에서 이전 버텍스 쉐이더의 명령어 구조에 추가하여 설계를 할 경우 그림 4 처럼 기존의 64비트에서 최소 109비트로 확장된 구조를 요구하게 된다.

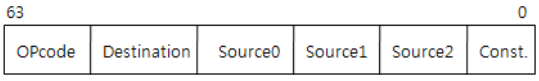

그림 1. 고정형 SIMD 64bit 명령어 형식
Fig. 1. The Format of Fixed SIMD 64bit Instruction

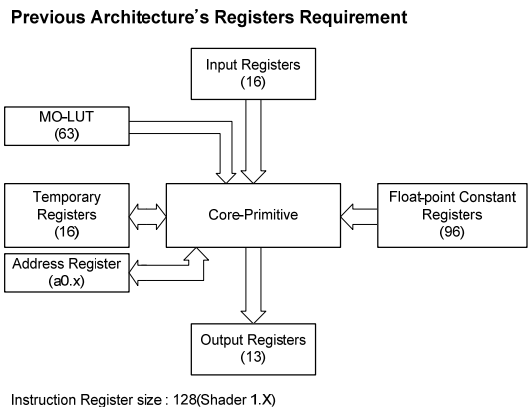

그림 2. OpenGL 1.0 API 레지스터 구조
Fig. 2. The Structure of OpenGL 1.0 API registers

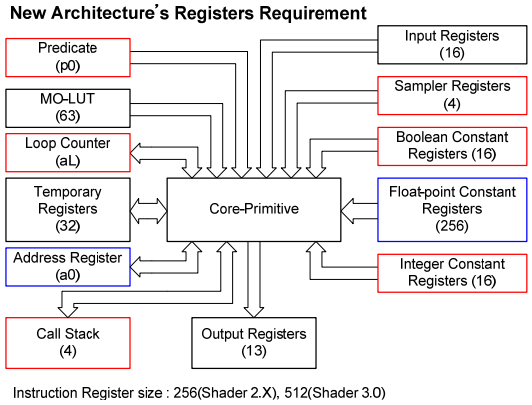

그림 3. Shader 3.0 API 레지스터 구조
Fig. 3. The Structure of Shader 3.0 API registers

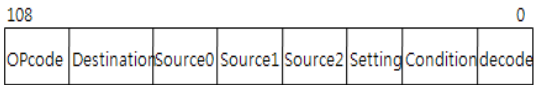

그림 4. Shader 3.0 API 구조를 반영한 확장된 명령어 형식
Fig. 4. The Extended Instuction Format based on Shader 3.0 API

이러한 확장된 구조는 실제 사용되는 명령어 필드(field)에 비하여 낭비되는 필드가 더 많아지며 그만큼 명령어 표현과 구동 효율이 떨어지게 된다. 이를 해결하기 위해서는 기존 명령어 구조의 설계 원인을 파악해야 한다.

사실상 기존의 명령어 구조는 모든 바탕을 API의 구조에 따르고 있다. 여기서 잘못된 점은 API의 지원을 위하여 하드웨어가 똑같은 구조로 지원할 필요가 없다는 것이다. 이는 A와 B가 하는 일을 수행하기 위해서는 반드시 A와 B가 동시에 필요한 것이 아니라 일반적인 사용이 가능한 또 다른 C로써 압축하여 실행 할 수도 있다는 것을 의미한다. 그림 3과 같이 추가된 레지스터 그룹들은 API의 데이터 관리 목적으로 선언되어 구분한 것이다. 이를 하드웨어에서도 똑같이 구분할 필요는 없다. 이미 API에서 먼저 구분하여 명령어로써 정의되기 때문에 모든 레지스터 그룹의 요건에 맞춘 기본적인 데이터 그룹을 설계하는 것으로써 그 그룹들의 역할을 충족할 수 있다. 본 연구에서 통합된 레지스터 그룹을 GPRs(General Purpose Registers)라고 명한다. 수개의 레지스터 그룹들을 단일의 GPRs로 통합함으로써 명령어 구조와 코어 구조는 좀 더 단순하게 될 수 있다.

2.2 제안하는 명령어 구조

제안하는 쉐이더 유닛은 두 개의 페이즈로 나누어 연산을 처리한다. 명령어 또한 1개의 명령어 구조로 이루어지지 않고 다중 명령어 구조를 채택해야 한다. 그림 5와 같이 기존의 Shader 명령어 형식을 따르지 않고 최대 4개까지 조합 할 수 있는 유닛 명령어(Unit Instruction) 구조를 채택하고 있다.

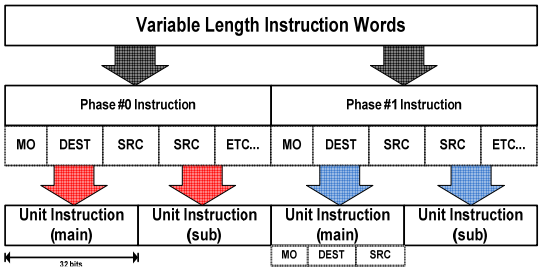

그림 5. 제안하는 명령어 형식
Fig. 5. The Proposed Instruction Format

간단히 말하면, 제안하는 가변길이 명령어는 유닛 명령어로 조합하여 만들어지는 명령어 이다. 하나의 제안하는 명령어는 앞서 말한 듀얼 페이즈에 맞추기 위하여 두 개의 페이즈로 나뉘어져 있다. 각 페이즈는 두 개의 SRC(source)와 한 개의 DEST(destination)을 가지고 해당

MO (Micro Operation) 처리하는 3 Operand 형태를 가진다. 기본적으로 두 개의 페이즈로 나뉘어 명령어를 구현할 수 있어서 듀얼 페이즈를 효과적으로 사용할 수 있다. 이렇게 두 개의 연산을 처리하는 명령어 구조로 인해 MIMD 프로세서 형식을 띄고 있다. 또한 각 페이즈 명령어는 두 개의 유닛 명령어로 만들어진다. 즉 유닛 명령어가 가변길이 명령어의 기본이 되는 것이다. 그림 6과 같이 하나의 유닛 명령어는 32bit로 이루어져 있으며 하나의 SRC와 한 개의 DEST를 가지는 2 Operand 형식의 명령어로 형태를 가진다. 유닛 명령어 하나로도 명령어 수행을 할 수 있고 유닛 명령어를 조합 방법에 따라 다양한 명령어를 구현할 수 있다.

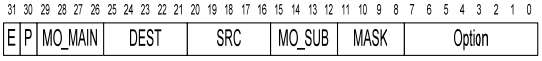

그림 6. 32비트 단위의 명령어 구조
Fig. 6. The Instruction Format composed of 32bit units

유닛 명령어를 가변적인 개수로 조합을 하기 때문에 조합 할 명령어 개수를 구분 하는 방법이 필요하다. 유닛 명령어에서 32번째 비트인 'E'(End of Instruction) 필드는 1 일 때 마지막 명령어를 뜻하는 필드이다. 이 'E' 필드를 이용하여 하나의 명령어가 이루는 유닛 명령어의 개수를 그림 7과 같이 1개에서 4개까지 조합을 할 수 있다. 단 명령어 조합 개수가 최대 4개이기 때문에 4번째 명령어는 'E' 필드를 체크하지 않는다.

'P'(Phase) 필드는 유닛 명령어가 처리가 되는 페이즈를 의미한다. 0인 경우는 페이즈 #0에 1인 경우는 페이즈 #1에서 처리 된다. 이 필드를 'E' 필드와 같이 비교하여 그림 24와 같이 8가지 방법으로 유닛 명령어를 조합 할 수 있다. 페이즈 #1의 유닛 명령어는 반드시 페이즈 #0의 명령어 뒤에 존재하여야 하며 각 페이즈는 최대 2개의 유닛 명령어를 선언할 수 있다.

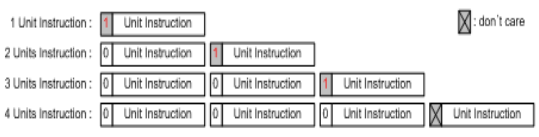

그림 7. 유닛 명령어를 이용한 명령어 조합
Fig. 7. The Instruction combination using Unit Instructions

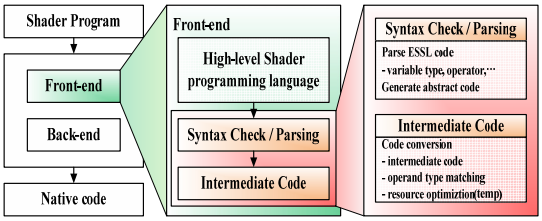

그림 8. 전단 컴파일러
Fig. 8. Front-end compiler

Ⅲ. 컴파일러 설계

3.1 제안하는 쉐이더 컴파일 구조

본 논문에서 구현된 컴파일 구조는 아래 그림 8과 같으며 작성된 셰이더 프로그램은 C와 유사한 형식 언어인 ESSL 문법의 쉐이더 프로그램으로 전단(Front-end) 컴파일러에 의하여 중간 코드를 생성하게 되고 이후 후단(Back-end) 컴파일러에 의하여 하드웨어 쉐이더 프로세서에서 동작 가능한 타겟 코드로 변환된다.

전단 컴파일러는 ESSL 문법 검사, 사용된 프로그램 변수 정보, 기타 Built-in 함수등의 파싱 정보등을 생성하고 이를 바탕으로 중간 코드를 생성한다. 이러한 중간 코드는 셰이더 프로그램 코드의 성격과 ANSI C 형식의 어셈블리 코드의 성격을 가지는 방식으로 구성되어 플랫폼 목적 코드 생성에 유리하다. 중간 코드는 OpenGL ARB Extension에서 사용되는 ARB v1.0 기준의 셰이더 어셈블리 코드형식을 가지도록 하며 Temporary, Constant, Vertex Input/Output, Fragment Input/Output 등의 셰이더 자원을 오퍼런드로 할당하여 표현된다.

3.2 추상 코드 생성기

컴파일러는 고급언어를 인식하는 단위의 토큰으로 분석하는 의미 분석기와 하위언어인 어셈블러가 인식할 수 있는 언어로 변환하는 추상 코드 생성기를 통해 레지스터가 할당 되지 않은 추상 코드를 생성하게 된다.

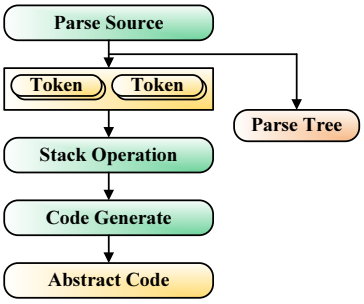

그림 9. ESSL 문법 검사 / 추상 코드 생성 과정
Fig. 9. The Process of ESSL grammar verification/ abstract code generation

1) 의미 분석기

3D Labs의 전단을 참조하여 제작 하였다. OpenGL ES 2.0 Spec에 BNF으로 제작된 고급 언어 문법을 Lex & Yacc을 이용하여 문법에 맞는 추상 구문 생성기를 제작한다.

2) 스택 연산

의미가 분석된 토큰들은 각 심볼 테이블에 저장되고 후위 연산 방식에 의해 연산자를 선두로 각 심볼 테이블들로부터 전송되는 토큰들을 스택연산에 사용된다. 각 심볼 테이블로 저장된 토큰들은 연산자와 피연산자로 구분되며 우선순위를 나타내는 깊이 값을 갖게 된다. 이 깊이를 이용하여 토큰들은 파스트리를 형성하게 된다. 이 파스트리와 스택의 최상이 노드와의 깊이(우선순위)를 비교하여 Push와 Pop연산을 결정하게 된다.

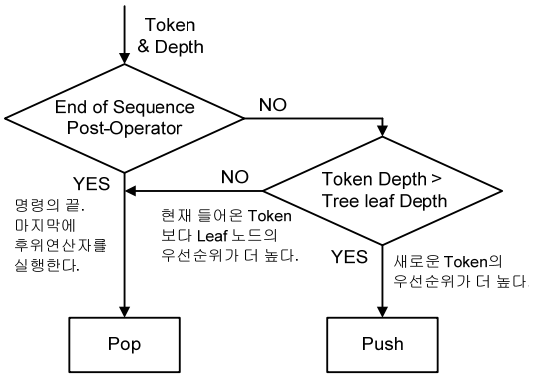

그림 10. 스택 연산 그래프
Fig. 10. The Stack operation graph

파스트리에서 전송되어야 할 노드보다 스택의 상위 연산자가 깊이(우선순위)가 높다면 Pop하여 추상코드 생성기로 전송하게 된다. 반대로 파스트리 노드의 깊이(우선순위)가 더 높다면 파스트리의 노드를 전송 받아 스택에 Push하여 추상코드 생성을 대기하게 된다.

3) 추상 코드 생성기

스택으로부터 전송되어온 토큰들은 연산자와 피연산자로 분류된다. 피연산자는 레지스터 할당을 위한 데이터로 변환하여 저장하며 연산자는 해당 연산자에 의해 저급언어의 명령어 셋을 선택한다. 할당 연사자가 포함되지 않은 모든 연산의 결과는 임시 레지스터에 저장하며 결과 레지스터를 다시 스택에 Push한다. 할당 연산자가 포함된 연산자는 결과 레지스터를 스택에 다시 Push한다. 다음으로 선택된 저급언어 명령어셋을 저장된 피연산자를 이용하여 레지스터가 할당되지 않은 추상코드를 생성한다. 생성된 추상코드는 다음 단계를 통해 메모리에 저장된다.

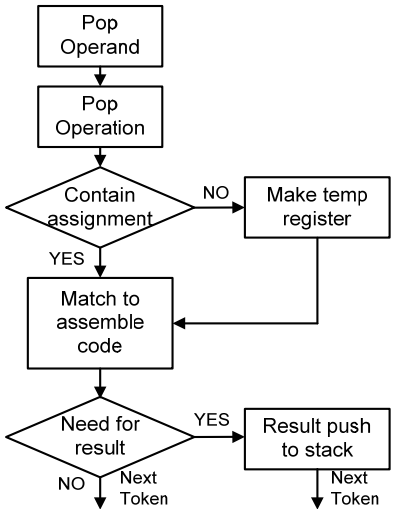

그림 11. 추상 코드 생성기
Fig. 11. The Abstract code generator

3.3 중간 코드 생성

고급 쉐이더 소스코드는 3D Labs에서 제공하는 FrontEnd와 추상 코드 생성기에 의해 추상 코드로 변환된다. 레지스터 할당은 추상 코드에 레지스터 배정하

는 작업이다. 레지스터 할당을 마친 추상 코드는 쉐이더 프로세서에서 처리될 타겟 실행 코드로 변환하는 작업을 하게 된다. 레지스터 할당을 통해 자원을 최대한 활용한다.

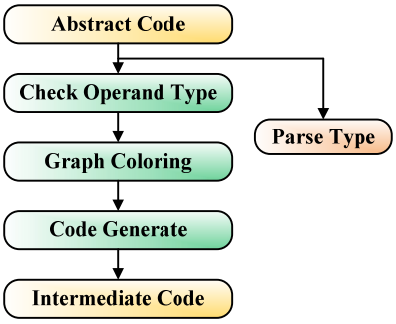

그림 12. 레지스터 할당 / 중간 코드 생성 과정
Fig. 12. The Process of Register allocation/ Intermediate code generation

Ⅳ. 연구 결과 및 분석

그림 13은 크로노스 그룹에서 제공한 OpenGL ES 2.0 API를 지원하는 기본적인 쉐이더 Conformance 테스트를 어셈블리 명령어로 구현하여 비교한 그림이다.

기존 명령어(Canonical Instruction)의 명령어 개수는 GP-GPU(General Purpose-Graphics Process Unit) 구조의 쉐이더에 포함된 모든 명령어를 지원하는 하나의 연산기가 있을 때, 하나의 스테이지에 한번의 연산만 처리하는 일반 구조일 경우 구현 되는 명령어 개수를 나타낸다. 제안하는 듀얼 페이즈 구조의 가변 길이 명령어(Proposed Instruction)는 서로 다른 연산을 동시에 처리 할 경우 구현된 명령어 개수를 나타낸다. 제안하는 명령어는 서로 다른 연산만 처리가 가능하기 때문에 각 기능마다 줄어드는 명령어 스테이지 수가 다르다. 특히 Radians 같은 기능은 성능을 반 이상 향상시킬 수 있고 Normalize 기능은 26%의 정도만 향상이 된다.

그림 13에서 전체 Conformance Test에서 제공하는 기본 GL 쉐이더의 기능 16개를 비교하여 보았을 때 전체 평균 37%가 줄어드는 것을 알 수 있다. 이는 앞서 설명한 바와 같이 수천~수만 개의 정점, 픽셀 데이터를 똑같은 명령어로 처리하기 때문에 전체 성능에는 특히 많은 영향을 끼친다.

또한 ALL, CEIL과 같은 개수의 명령어들을 서로 비교하면 가변길이 명령어를 사용하기 때문에 얼마나 효율적으로 명령어를 구현하느냐에 따라 줄일 수 있는 명령어 개수가 다르기 때문에, 명령어를 효과적인 조합으로 성능을 더욱 향상 시킬 수 있는 가능성도 존재한다.

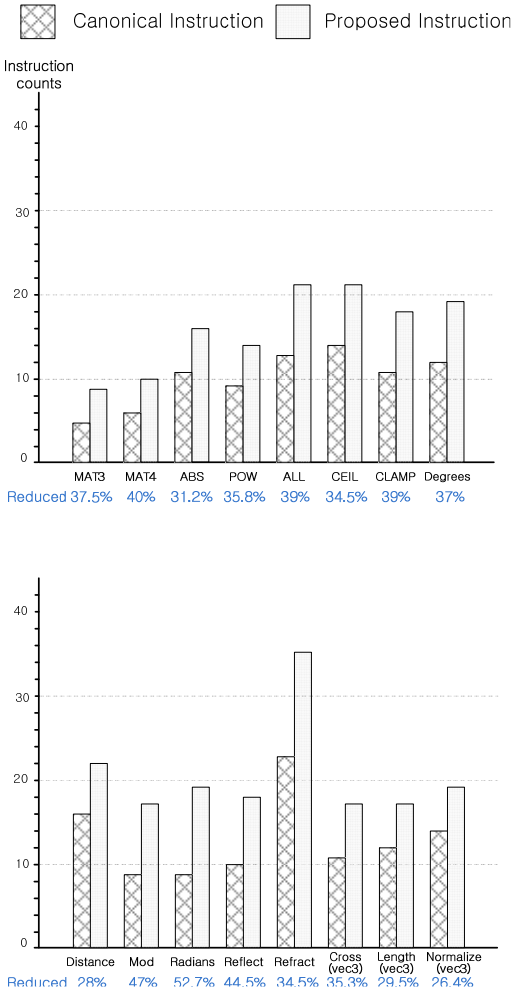

그림 13. 명령어 conformance 테스트 결과
Fig. 13. The results of Instruction conformance test

V. 결 론

본 논문에서는 휴대 정보 기기 환경에서의 요구사항을 만족시킬 수 있는 완전 프로그램이 가능한 GP-GPU 구조의 쉐이더 설계에 대해 기술하였다. 본 논문에서 설계된 쉐이더 유닛은 가변길이 명령어를 이용하여 OpenGL ES 2.0 의 기능을 모두 지원할 수 있도록 하였고 듀얼 페이즈 구조를 사용하여 칩 면적을 늘리지 않고 성능을 평균 37% 향상 시킬 수 있었다. 또한 본 쉐이더는 면적대비 성능 면에서 기존의 다른 프로세서에 비해 월등한 성능을 가지고 있다. 그리고 최대 400MHz(65nm)의 동작을 보장하며, 본 논문의 쉐이더 유닛과 유사한 기능을 가지는 ARM의 Mali 면적[4]과 다른 연구[5][6]에 비해 1/4 면적을 가진다.

또한, 본 논문에서는 가변길이 명령어를 지원하는 쉐이더 컴파일러를 새롭게 설계 구현하였다. 쉐이더 컴파일러는 ESSL 수준에서 작성된 사용자 프로그램을 지원하며 OpenGL ES 2.0 API와 호환성을 갖는다.

본 논문에서는 쉐이더 명령어와 ESSL 컴파일러의 OpenGL ES 2.0을 지원여부를 검증하기 위해 크로노스 그룹에서 제공하는 그래픽스 파이프라인을 명령어로 구현하여 검증하였다.

또한, 본 논문에서 개발된 쉐이더 IP는 현재 ARM사의 Mali 프로세서가 선점하고 있는 모바일 그래픽 프로세서에 대응하는 기술로 평가된다.

참고문헌

[1] Microsoft Shader3.0, http://msdn.microsoft.com

[2] H.K. Jeong, "Design of 3D Graphics Geometry Accelerator using the Programmable Vertex Shader" ITC-CSCC 2006.

[3] Mauricio Breternitz, Jr., "Compilation, Architectural Support, and Evaluation of SIMD Graphics Pipeline Programs on a GP-CPU" Proc. of the 12th int'l conference on parallel architectures and compilation techniques.

[4] ARM http://www.arm.com/

[5] J.H Sohn, et al., "A 155-mW 50-Mvertices/s Graphics Processor With Fixed-Point Programmable Vertex shader for Mobile Applications", IEEE J.l of Solid State Circuits, Vol 41, No. 5, pp.1081-1091, May, 2006

[6] Jeong-Ho Woo, et al. "A 195mW, 9.1MVertices/s fully programmable 3D graphics processor for low power mobile device", Solid-State Circuits Conference, 2007

저자소개

곽재창(Jae-Chang Kwak)

1989 Univ. of Iowa 전산학석사
1993 Univ. of Iowa 전산학박사
1995~현재 서경대학교 컴퓨터과학과 교수

※관심분야 : 네트워크프로토콜, QoS, Scheduing, 멀티미디어 및 컴퓨터 그래픽스

박태룡(Tae-Ryoung Park)

1987 한양대학교 수학과 이학석사
1995 한양대학교 수학과 이학박사
1994~현재 서경대학교 소프트웨어/컴퓨터공학과 교수

※관심분야 : 정보보호 및 암호 알고리즘, 멀티미디어 및 컴퓨터 그래픽스