임베디드 그래픽 프로세서를 위한 OpenGL ES 컴파일러 개발

임수준○ 송준섭 신동군
성균관대학교 정보통신공학부
lang33@skku.edu song0319@skku.edu dongkun@skku.edu

OpenGL ES Compiler Implementation for Embedded Graphic Processor

Soojun Im○ Junsup Song Dongkun Shin
Sungkyunkwan University

요 약

오늘날 휴대용 기기에서의 그래픽 처리 요구사항이 증가함에 따라 저전력, 저비용 그래픽 프로세서의 필요성이 대두되고 있다. 이에 따라 크로노스 그룹은 휴대기기를 위한 그래픽 API 표준인 OpenGL ES 2.0을 발표하였다. 본 논문에서는 OpenGL ES 2.0을 상정하여 구성된 그래픽 프로세서를 위한 쉐이더 컴파일러를 개발하고 최적화하는 연구를 수행하였다. 개발된 컴파일러는 OpenGL ESSL로 작성된 쉐이더 프로그램을 정상적으로 컴파일하고 동작시켰으며 타겟 GPU에 적합한 최적화 기법을 적용하여 쉐이더 프로그램의 크기를 최대 10%가량 절감하고 성능을 10~15%가량 향상시켰다.

1. 서 론

오늘날 스마트폰, 타블렛PC, PMP, 휴대용 게임기 등의 임베디드 기기에서도 고 수준의 영상 처리가 필요하게 됨에 따라 데스크탑 PC에서만 사용되던 GPU(Graphic Processing Unit)가 휴대기기에도 주요한 연산 장치로 사용되기 시작하였다. 다양한 그래픽 처리 기법이 요구되면서 기존의 고정 파이프라인 구조로부터 사용자가 직접 프로그래밍하여 원하는 영상 처리를 수행할 수 있는 프로그래밍 가능한 쉐이더(Programmable shader)의 지원이 요구되었다. 이러한 쉐이더를 작성하고 GPU에서 수행하기 위해서 크로노스 그룹에서는 기존의 그래픽 처리 표준 API로 구성된 OpenGL을 임베디드 기기에 알맞게 수정한 OpenGL ES API[1]와 ESSL(Embedded System Shader Language)을 발표하였다. 현재 OpenGL ES 는 2.0 버전까지 발표된 상태이다.

OpenGL ES 2.0은 임베디드 기기 환경에서 3D 그래픽을 처리하기 위해 제공되는 API이다. OpenGL ES 2.0은 그래픽처리 파이프라인으로 구성되어 있으며 입력된 정점 정보들은 각 스테이지를 거쳐 최종적으로 화면에 출력되는 내용을 저장하는 프레임 버퍼에 기록된다. 이러한 그래픽 파이프라인 스테이지는 다음과 같이 구성된다. 1) 정점 정보를 입력받아 이를 3차원 공간에서의 도형으로 구성하는 버텍스 쉐이더를 실행하고 2) 이러한 도형을 2차원 이미지로 변환하는 주사선 변환을 거친 다음에 3) 텍스쳐 메모리로부터 텍스쳐 데이터를 입력받아 생성된 2D 이미지에 픽셀 별 연산을 수행하는 프래그먼트 쉐이더를 거쳐 프레임 버퍼에 실제 화면을 출력한다.

이 과정에서 버텍스 쉐이더와 프래그먼트 쉐이더의 내용을 사용자가 원하는 작업으로 프로그래밍 할 수 있다. 이러한 쉐이더 프로그램은 ESSL로 작성된다. ESSL은 C와 유사한 문법을 지니고 있으나 데이터 형이나 벡터 연산자 등 데이터를 다루는데 있어 일반적인 C와 차이점을 가지고 있다. 데이터 형은 일반적인 void, bool, int, float 외에 vector(vec2, vec3, vec4 등)와 matrix(mat2, mat3, mat4) 와 같이 다수의 변수를 벡터 및 행렬로 구성하여 사용할 수 있다. 또한 각각의 변수에 attribute, varying, uniform 등의 수식어를 사용하여 변수가 어떤 과정에서 사용되는 입출력 변수인지를 나타내고 이에 따라 물리적인 저장 공간이 결정된다. 그 외에 텍스쳐 데이터를 저장하고 있는 sampler 형의 변수가 존재하며 포인터 형의 변수는 사용할 수 없다.

ESSL로 작성된 쉐이더 프로그램은 쉐이더 컴파일러에 의해서 이진 코드로 변환되어 GPU에서 실행된다. 기존의 CPU를 대상으로 한 프로그램과 달리 쉐이더 프로그램은 연산 명령어나 변수가 할당되는 메모리가 다른 특성이 있기 때문에 이를 고려한 컴파일러 기법이 필요하다. 기존의 ESSL 컴파일러의 연구[2][3] 에서는 이러한 GPU의 특성이 충분히 고려되지 않았다.

본 논문에서는 위에서 설명한 OpenGL ES 2.0을 위한 컴파일러를 개발하였다. 해당 컴파일러는 타겟 GPU의 특수한 구조에 맞추어 구현되었으며 이를 효율적으로 사용할 수 있도록 다양한 최적화 기법을 적용하였다. 논문의 내용은 다음과 같이 구성된다. 2장에서는 컴파일러를 구현한 타겟 GPU에 대하여 설명하고 3장에서는 타겟 GPU에 맞게 컴파일러를 구현한 내용을 설명한다. 4장에서는 실제 OpenGL ES 2.0 예제를 사용한 테스트를 수행하여 각 최적화 기법의 효과를 평가하였으며 마지막으로 5장에서 결론을 보이고 있다.

2. 타겟 GPU 구조

본 논문에서 사용한 타겟 GPU[4]는 임베디드 기기를 위해 개발된 프로그래밍 가능한 쉐이더 프로세서이다. 임베디드 기기에서 사용되는 것을 전제로 저전력, 저비용을 달성하기 위해서 프로세서 내부에 하나의 ALU(Arithmetic-Logical Unit)만을 사용하고 있다. ALU는 그래픽 스트림 데이터를 병렬적으로 처리하기 위하여 벡터 데이터의 각 컴포넌트(x, y, z, w)를 동시에 처리할 수 있도록 만들어진 연산 장치와 특수 함수를 연산하기 위한 SPU(Special Functional Unit)로 구성된다. 그러므로 일반 스칼라 연산을 위한 ALU보다 효율적으로 그래픽 스트림 데이터를 처리할 수 있다.

또한 타겟 GPU는 하나의 ALU를 최대한 활용하여 성능을 향상시키고자 VLIW(Very Long Instruction Word)기법을 활용한 듀얼 페이즈 명령어를 사용한다. 듀얼 페이즈 명령어는 주 명령어와 보조 명령어로 나누어지며 각각의 명령어는 2개의 유

본 연구는 지식경제부 및 한국산업기술평가관리원의 산업융합원천기술개발사업(정보통신)의 일환으로 수행하였음. [KI0018-10041244, 스마트TV 2.0 소프트웨어 플랫폼]

닛으로 구성된다. 각 유닛은 32비트의 명령어 필드를 가지며 명령어에 따라 필요한 정보가 1개의 유닛으로 충분한 경우에는 1개의 유닛을 사용하고 그렇지 않은 경우에는 2개의 유닛을 모두 사용하여 명령어를 표현한다. 주 명령어와 보조 명령어는 별도의 독립된 명령어로써 ALU에서 같은 종류의 연산을 하지 않는 경우에는 동시에 실행할 수 있기 때문에 한 사이클에 최대 2개의 명령어를 수행할 수 있다. 그러나 이러한 하드웨어의 특성을 활용하기 위해서는 컴파일러에서 적절한 듀얼 페이즈 명령어를 생성해주어야할 필요가 있다

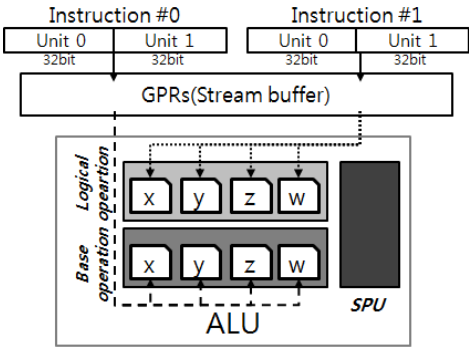

그림 1. 타겟 GPU 명령어 구조

타겟 GPU가 사용하는 메모리는 그림 2에서 보는 것과 같이 instruction bank, global memory, stream buffer, sampler memory가 있다. 각각의 메모리 공간에는 쉐이더 프로그램에서 선언된 변수들이 저장된다. instruction bank에는 수행할 쉐이더 코드가 저장되며 각 쓰레드들이 공유하는 global memory는 uniform으로 선언된 변수와 컴파일 결과 생성된 상수 값이 저장된다. 각 쓰레드별로 할당되는 stream buffer는 각 쉐이더가 사용하는 임시 변수를 위한 레지스터와 입출력 값들의 경로이다. 마지막으로 texture memory에는 텍스쳐/샘플러 데이터가 저장된다. 컴파일러는 보다 효율적으로 쉐이더 프로그램을 수행할 수 있도록 제한된 메모리 공간을 활용하여야 한다.

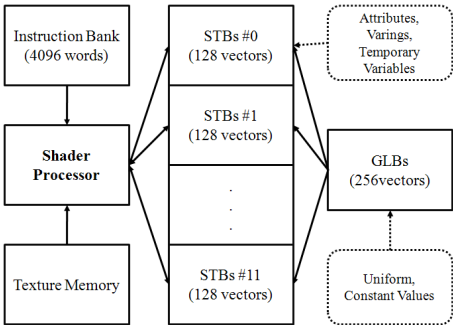

그림 2. 타겟 GPU 메모리 구조

3. 컴파일러 구현

3.1 컴파일러 개요

OpenGL ES 2.0의 버텍스 쉐이더와 프래그먼트 쉐이더는 사용자에 의해 작성되는 프로그래밍 가능한 쉐이더로써 OpenGL ES 컴파일러에 의해 컴파일되어 그래픽 파이프라인에서 동작한다. 본 논문에서 개발한 컴파일러는 3DLabs의 ESSL 전반부(Front-end) 컴파일러를 사용한 전처리기, 스캐너, 파서를 사용하고 컴파일러와 코드 생성기를 타겟 GPU에 맞게 새로 구현하였다. 컴파일러에서는 해석된 중간 코드를 사용하여 어셈블리 코드를 생성하고 코드 생성기에서는 생성된 어셈블리 코드를 기계어로 변환한다.

3.2 프로시저를 고려한 전역 레지스터 할당 기법

어셈블리 코드를 기계어로 변환할 때 변수들은 각각의 데이터 형에 따라 타겟 GPU의 물리 메모리에 할당된다. 특히 stream buffer에 할당되는 임시 변수는 각 쓰레드마다 다른 값을 가지며 프로그램이 수행되는 동안에 읽고 쓰기가 동시에 발생하고 다양한 생존 주기를 가지고 있기 때문에 이를 적절히 할당하는 것이 성능에 큰 영향을 미친다.

본 논문에서는 임시 변수를 할당하기 위해서 그래프 컬러링 기법을 사용하였다. CPU를 사용하는 프로그램은 임시 변수들은 각각의 스코프에 따라 프로시저 내부에서 레지스터를 할당받고 도중에 다른 프로시저가 호출되는 경우에는 그 값들을 스택 메모리에 저장하는 방식으로 레지스터를 할당한다. 그러나 타겟 GPU를 비롯한 일반적인 GPU 환경에서는 각 프로세스의 스택 메모리로 사용할 메모리가 없기 때문에 위와 같은 레지스터 스필링 기법을 사용할 수 없다. 이 때문에 프로시저 내부 스코프만을 고려하는 것이 아닌 전역적인 레지스터 할당 방법이 필요하다.

이러한 제약사항을 고려하여 레지스터를 할당하기 위하여 본 논문에서는 다음과 같은 방법을 사용하였다. 각 함수는 마치 인라인 함수인 것처럼 간주되어 변수들의 전체 프로그램 스코프에서의 생존 주기를 분석한다. 이를 기반으로 함수들의 호출 그래프를 생성하여 가장 호출 깊이가 깊은 함수에서 정의된 변수부터 레지스터를 할당한다. 다음 단계에서 다른 함수의 변수에 레지스터를 할당할 때는 기존에 할당된 레지스터와 생존 주기가 겹치지 않는 경우에만 레지스터를 재사용할 수 있다. 이와 같은 방법으로 컴파일러는 레지스터 스필링 없이 다양한 함수가 호출되는 상황에서의 레지스터 할당을 수행할 수 있다.

또한 OpenGL ES 2.0 표준에서 global memory를 최대한 활용하기 위한 방안으로 제안된 uniform packing 기법을 적용하여 uniform/constant 변수들을 할당하는 것으로 global memory를 최대한 효율적으로 사용하였다.

3.3 하위 레지스터를 이용한 코드 크기 최적화

2장에서 설명한 것과 같이 stream buffer는 임시 변수와 입출력 변수들을 저장하는데 사용된다. 타겟 GPU는 stream buffer를 접근하기 위한 주소 값을 하위 5비트와 상위 2비트로 나누어서 사용한다. 이러한 환경에서 Mov와 같이 피연산자 2개인 명령어의 두 피연산자가 모두 0~31까지 하위 32개의 stream buffer에 할당된 경우에는 1개의 유닛으로 명령어를 표현 가능하다. 그러므로 이러한 명령어의 피연산자를 stream buffer의 하위 32개에서 할당하는 것으로 코드 크기를 줄일 수 있다. 이를 위해서 레지스터 할당을 두 단계로 나누고 먼저 가상의 레지스터 번호에 레지스터를 할당한 다음에 참조 횟수가 가장 많은 명령어부터 하위 32개에 순차적으로 배치한다. 이러한 기법을 사용하여 1개의 유닛 명령어를 최대한 활용하는 것으로 코드 크기를 줄일 수 있다.

3.4 듀얼 페이즈 명령어를 사용한 실행 시간 최적화

2장에서 설명하였던 것과 같이 타겟 GPU는 한 사이클에 최대 2개의 명령어를 동시에 수행하는 것이 가능하다. 그러나 하나의 ALU에서 같은 종류의 연산을 동시에 수행하는 것은 불가능하다. 예를 들어 두 개의 명령어가 동시에 Add와 Sub 명령어를 수행할 수는 없다. 이러한 제약 사항에 맞추어 컴파일러는 적합한 2개의 명령어를 선정하여 듀얼 페이즈 명령어를 구성해주어야 한다. 본 논문에서는 연속된 두 개의 명령어가 다른 종류의 연산인 경우에 하나의 듀얼 페이즈를 생성한다 또한 베이직 블록 내부에서 같은 종류의 연산이 연속된 경우에

실행 결과가 달라지지 않는다면 명령어의 위치를 재배치하여 최대한 듀얼 페이즈 명령어의 특성을 활용할 수 있도록 한다.

4. 실험 및 검증

본 논문에서 구현한 컴파일러를 검증하기 위하여 타겟 하드웨어의 3D IP 테스트 벤치를 사용하였다. 테스트 벤치는 OpenGL ES 2.0 표준을 따라 구성되어 쉐이더 API 및 하드웨어 모델링을 따라 동작한다. 컴파일러는 테스트 벤치의 모듈로서 버텍스 쉐이더와 프래그먼트 쉐이더를 수행할 때 OpenGL 쉐이더 API에 의하여 호출되어 컴파일을 수행하고 생성된 결과물인 바이너리 코드와 입출력 데이터 스트림 정보, uniform/constant의 값과 주소 정보를 반환한다. 그래픽 파이프라인은 API를 통하여 이러한 정보를 접근하고 주어진 쉐이더 코드를 수행하여 출력한다.

실험에 사용된 예제는 OpenGL ES Conformance Testing[5]에서 disable, light, texture의 3가지 소스 코드를 사용하였다. 그림3에서 볼 수 있는 것처럼 컴파일된 쉐이더 코드는 그래픽 파이프라인을 따라 정상적으로 3D 그래픽 화면을 출력하는 것을 확인하였다.

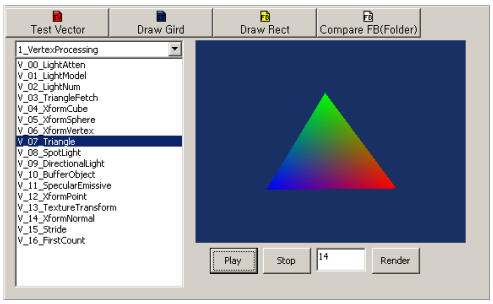

그림 3. Traingle 예제

3.3절에서 제안한 하위 레지스터를 이용한 코드 크기 최적화 기법의 효과를 실험하였다. 그림 4에서 볼 수 있는 것과 같이 할당되는 변수의 순서대로 레지스터를 할당하는 단순한 기법에 비하여 유닛 크기를 고려한 레지스터 할당 최적화 기법을 적용한 결과 평균 4.7%가량 코드 크기가 줄어드는 것을 확인하였다. Disable 쉐이더 코드의 프래그먼트 쉐이더에서는 사용하는 변수의 개수가 적어 모든 레지스터가 하위 레지스터에 할당되기 때문에 해당 최적화 기법이 효과를 발휘하지 못한 반면 Mov나 Global load와 같은 1유닛 명령어의 비중이 많은 텍스쳐 쉐이더 코드의 프래그먼트 쉐이더에서는 10.3%가량 코드 크기가 줄어드는 것을 확인하였다.

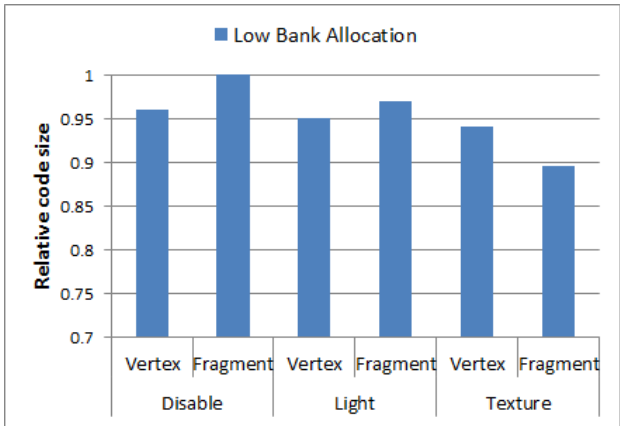

그림 4. 하위 레지스터 할당 기법의 효과

3.4 절에서 소개한 듀얼 페이즈 명령어 생성 기법을 싱글 페이즈 명령어와 비교하였다. 그림 5에서 보는 것처럼 듀얼 페이즈 명령어를 사용하면 평균 10.7% 가량 수행속도가 향상되었다. 특히 점프나 주소 연산보다 계산을 수행하는 부분이 많은 texture 예제에서는 최대 16.8% 가량 실행 속도가 빨라지는 것을 볼 수 있다.

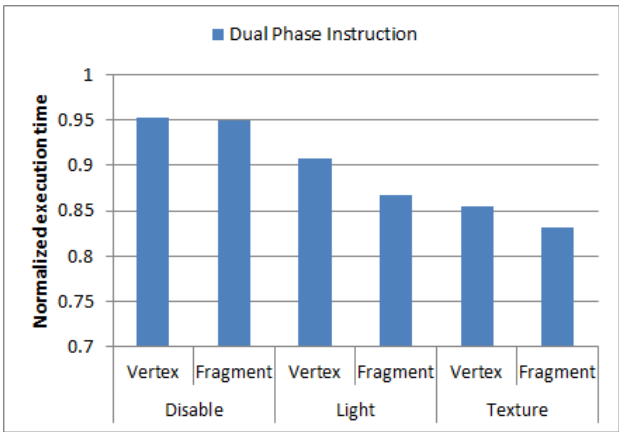

그림 5. 듀얼 페이즈 명령어 활용 효과

5. 결 론

본 논문에서는 OpenGL ES 2.0을 위한 쉐이더 프로그램을 위한 컴파일러를 구현하고 타겟 GPU에 적합한 최적화 기법을 도입하여 이를 실험을 통하여 검증하였다. 구현된 컴파일러는 다양한 쉐이더 프로그램과 입력 변수에 따라 쉐이더 프로그램을 동작시켜 3D 그래픽을 정상적으로 출력하였다. 또한 하위 레지스터 우선 할당 기법을 사용하여 쉐이더 프로그램의 코드 크기를 최대 10% 감소시켜 제한된 메모리를 효율적으로 사용할 수 있게 하였으며 명령어 재배치를 이용한 듀얼 페이즈 명령어로 쉐이더 프로그램 수행 시간을 최대 16.8% 가량 감소시켰다.

후속 연구로 타겟 GPU에서 프로그램 수행 속도를 향상시키기 위한 uniform/constant 할당 방법과 uniform/constant 변수를 stream buffer로 불러오는 연산을 최소화하는 레지스터 할당 기법 등을 개발할 계획이다.

참고문헌

[1] Khronos Group, OpenGL ES 2.0, http://www.khronos.org/opengles/

[2] NTNU, OpenGL ES Shading Language Compiler Project Report, www.idi.ntnu.no/emner/tdt4290/Rapporter/2005/oglesslc.pdf, 2005

[3] Robart, M. Hill, S., ESSL compiler for embedded 3D graphics architecture, 2009. ICCE '09, 10-14, Jan. 2009

[4] Woo-Young Kim, Bo-Haeng Lee, Kwang-Yeob Lee, Jae-Chang Kwak, Design of a fully programmable shader processor for low power mobile devices, TENCON '09, 23-26, Jan. 2009

[5] OpenGL ES Adoption and Conformance Testing, http://www.khronos.org/opengles/adopters/