메모리 전송 효율을 개선한 programmable Fragment 쉐이더 설계

박태룡*

A Design of Programmable Fragment Shader with Reduction of Memory Transfer Time

Tae-ryoung Park*

요 약

3D 그래픽을 처리하는 연산 과정에는 고정적인 연산만을 수행하는 영역과 Shader 등과 같은 명령어에 의한 프로그래밍이 요구되는 영역이 구분되어 있다. 이러한 3D 파이프라인의 특성을 고려하여 fixed 구조로 설계한 graphics hardware와 명령어 기반의 programmable hardware를 혼합한 구조로 설계하면 효율적인 그래픽 처리가 가능하다. 본 논문에서는 이러한 혼합 구조에 적합한 OpenGL ES(Open Graphics Library Embedded System) 2.0을 지원하는 Fragment Shader를 설계하였다. fixed hardware와 Shader간 데이터 입출력으로 인해 발생할 수 있는 전체 파이프라인의 지연을 줄일 수 있도록 내부 인터페이스를 최적화하였으며 Shader 내부 레지스터 그룹을 interleaved 구조로 설계하여 레지스터 면적과 처리 속도를 개선하였다.

ABSTRACT

Computation steps for 3D graphic processing consist of two stages - fixed operation stage and programming required stage. Using this characteristic of 3D pipeline, a hybrid structure between graphics hardware designed by fixed structure and programmable hardware based on instructions, can handle graphic processing more efficiently. In this paper, fragment Shader is designed under this hybrid structure. It also supports OpenGL ES 2.0. Interior interface is optimized to reduce the delay of entire pipeline, which may be occurred by data I/O between the fixed hardware and the Shader. Interior register group of the Shader is designed by an interleaved structure to improve the register space and processing speed.

키워드

3차원그래픽, 프래그먼트 쉐이더, OpenGL

key word

3D Graphics, Fragment Shader, OpenGL

* 서경대학교 컴퓨터공학과 교수(trpark@skuniv.ac.kr)

접수일자 : 2010. 06. 15
심사완료일자 : 2010. 09. 14

Ⅰ. 서 론

초기 그래픽 처리장치는 OpenGL ES 1.0을 지원하는 fixed pipeline 구조로 파이프라인의 고정된 기능만을 수행할 수 있어 표현 가능한 효과에 한계가 있다. 이를 보완하기 위해 개발된 programmable Shader는 fixed 구조에 비해 느리지만 연산 과정을 개발자가 프로그래밍이 가능하도록 지원하여 다양한 처리가 가능한 장점이 있다.[1] 최근 이러한 fixed hardware 구조와 programmable Shader의 특성을 반영하여 데이터 처리량이 많고 고정적인 연산을 수행하는 stage는 fixed hardware로 구현하고, 그래픽 효과를 높일 수 있는 Fragment Shading 기능을 programmable Shader로 구현하는 혼합구조가 주목받고 있다.[2]

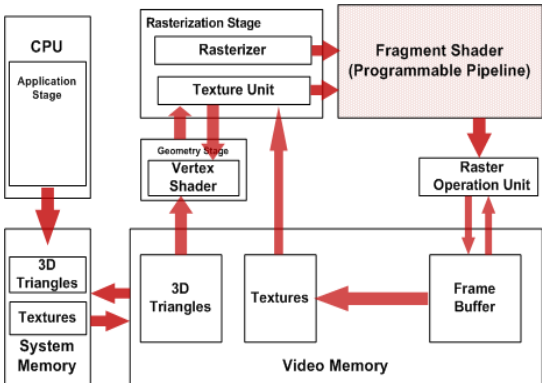

그림 1. 3D 그래픽 파이프라인 흐름도
Fig. 1 3D graphics pipeline flowchart

본 논문에서는 그림 1.과 같이 fixed graphics pipeline에 적용 가능한 OpenGL ES 2.0을 지원하는 programmable Fragment Shader를 제안한다. 기존 구조에서는 스레드의 동작을 지원하는 레지스터 파일을 dual port SRAM으로 구성하여 면적 및 코어의 최대 동작 속도가 제한적인 문제가 존재한다.[3] 또한 명령어 처리에 요구되는 데이터 입출력에 있어 여러 스레드가 버스 인터페이스를 공유하기 때문에 전송에 지연이 발생하여 파이프라인의 효율이 떨어지게 된다.[4]

이를 개선하기 위해 Shader 내부에 존재하는 범용 레지스터 그룹의 효율적인 통합 구조와 fixed hardware와 Shader 간의 데이터 입출력 가속 기법을 제안하고, 이를 fixed pipeline hardware의 Fragment Shader에 적용하여 그 성능을 검증하였다.

Ⅱ. Shader 구조 개선의 필요성

Programmable Shader는 병렬 처리 연산이 가능하도록 프로세스 코어를 스레드라는 처리 단위로 나누어 여러 stage로 나누어 동시에 처리하는 형태로 이루어져 있다. 프로세서는 스레드별로 데이터의 독립적인 처리를 위해 레지스터 그룹을 별도로 구성하게 되는데 이때 스레드의 개수가 많을수록 레지스터 그룹에 접근하는 과정에서 많은 딜레이가 발생하게 된다. 따라서 코어와 레지스터 그룹간의 인터페이스를 단순화하는 것이 중요한 이슈가 된다. 또한 fixed hardware와 병렬 처리 연산을 수행하는 Shader 사이에서는 픽셀 데이터의 입출력 시 필연적으로 지연이 발생하게 되는데 이는 전체 파이프라인 처리 성능에 영향을 미치게 된다. Fragment Shader는 픽셀 정보를 처리하기 위해 Fragment coordinate, Varying, point 등의 많은 데이터를 Shader 연산의 전처리 과정에서 요구하게 된다. Fragment Shader는 이러한 데이터 입력을 외부 메모리 전송 인터페이스로 받아들이게 되고, 출력 또한 동일한 형태로 처리하기 때문에 Shader 내부 레지스터 그룹과 외부 fixed hardware 간의 고속 입출력 인터페이스의 구현이 Shader의 유휴 시간을 감소시켜 픽셀 데이터의 입출력을 가속화 하게 된다.

Ⅲ. 기존 Fragment Shader 구조

본 논문에서 구현한 programmable Fragment Shader는 각 스레드별로 STBs(Stream Buffers)라는 레지스터 그룹을 구성하여 병렬처리를 지원하고 있다. 그림 2.는 기존의 STBs와 코어 및 외부 버스간의 읽기 쓰기 구조를 나타낸다.[3]

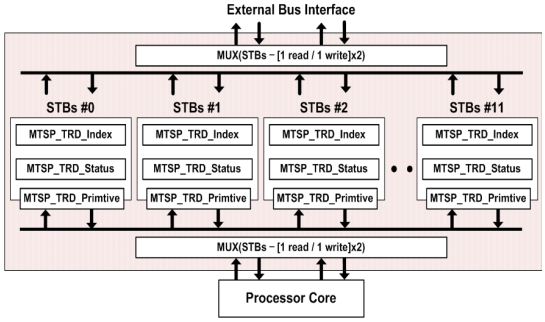

그림 2. 기존의 STBs 연결 구조
Fig. 2 Conventional structure of STBs

각 STBs는 4D vector * 32 * 64 크기만큼 할당되어 있으며 Shader 내부에는 총 12개의 스레드가 존재한다. 스레드 동작 시, 각 STBs는 프로세서의 코어와 직접 연결되어 현재 실행중인 스레드의 STBs를 선택하기 위해 매 클럭마다 muxing 과정을 거치게 된다. Shader 내부에서 수행되는 스레드의 개수가 많아질수록 MUX의 크기 역시 커져야 하며 이로 인해 발생하는 딜레이는 전체적인 처리 성능의 저하로 이어지게 된다.

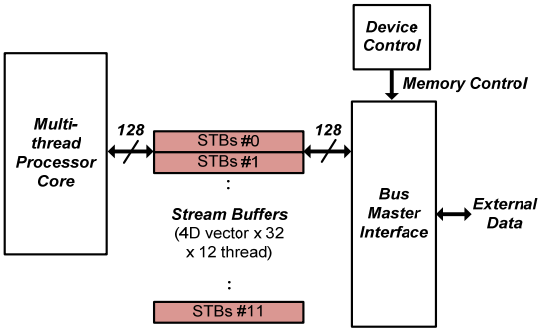

그림 3. 외부 메모리와 Shader 코어와의 연결 구조
Fig. 3 A linking structure between external memory and Shader

그림 3.은 기존 Shader에서의 데이터 입출력을 위한 STBs와 코어 및 외부 메모리 간의 인터페이스를 나타낸다.[4] Fragment Shader에서 STBs에 관련된 외부 입출력은 스레드 동작 이전과 픽셀 데이터에 대한 모든 처리가 끝난 시점에서 발생한다. 기존 구조에서의 외부 입출력은 내부 버스 인터페이스를 거쳐 한번에 최대 32bit x 4D vector size만큼만 전송이 가능하다. Fragment Shader는 Fragment 출력 데이터 Fragment coordinate, color, Front face를 연산하기 위해 최저 32bit * 4D * 10(FragCoord 1, Varying 4D 8, PointCoord 1)개의 Fragment 입력을 요구한다. 이에 따라 전처리 과정에서 Shader 동작을 위해 단일 스레드당 10 클럭을 소모하게 되며 Shader의 동작 시점은 이에 비례하여 지연된다. 또한 다중 스레드 구조를 채택하고 있는 현 프로세서에서는 입력에 있어 스레드 개수만큼 추가적인 지연이 발생하므로 이를 해결하기 위한 구조적 개선이 요구된다.

Ⅳ. 제안하는 Shader 입출력 구조

표 1.은 programmable Shader 파이프라인에서의 각 stage별 수행과정이다.

표 1. Shader 파이프라인 단계
Table. 1 Shader pipeline stage

Pipeline stage	Stage operation
TRD(Thread)	스레드 상태 및 명령어 주소 액세스
IA(Inst. address)	명령어 주소 발생
IF(Inst. fetch)	명령어 Fetch
IO(Inst. organization)	명령어 정렬 및 소스 어드레스 발생
OF(Operand fetch)#0~2	소스 어드레스 발생 및 오퍼란드 패치
ID(Inst. decode)#0~3	명령어 디코딩
EX(Execution)#0~2	명령어 연산 수행
WB(Write back)#0~1	연산 결과 저장

위의 16개의 stage 중에서 TRD, IA, IF, IO 단계의 경우 EX1, EX2, WB0, WB1의 실행과 같은 시간에 수행되어 4 stage-overlap되는 구조를 가지고 있다. OF, ID, EX, WB 과정은 코어의 특성에 따라 다수의 stage로 나누어 처리하게 된다.

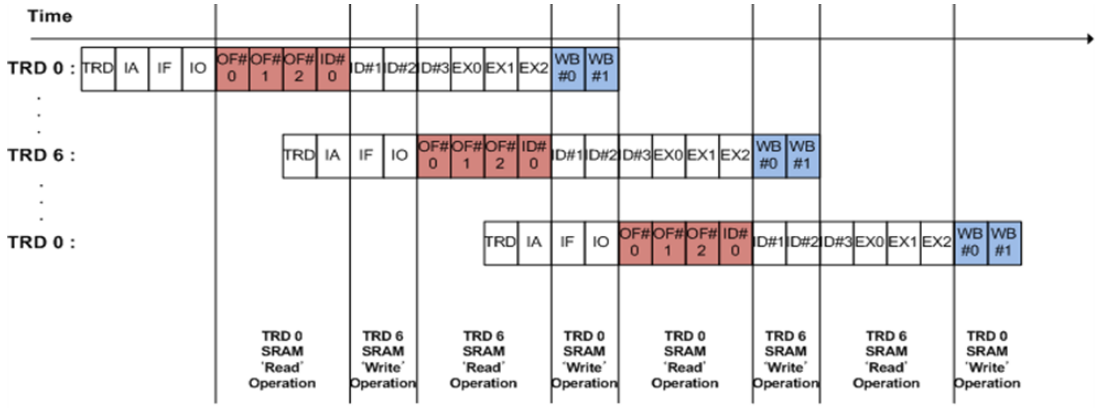

그림 4. 스레드 동작간 메모리 접근 분석
Fig. 4 Analyzing memory access of threads

그림 4.는 매 클럭당 스레드 내부에서 수행되는 파이프라인 stage를 나타내고 있다. 모든 명령어는 총 12개의 스레드에서 클럭 단위로 시간차를 두어 동시에 처리되며 각 stage는 명령어의 종류에 관계없이 동일하게 구성된다.[4]

하나의 스레드는 처리될 명령어의 디코딩 결과에 따라 수행될 ALU를 선택하게 되는 operand fetch 과정을 수행한다. 이 단계는 OF#0, OF#1, OF#2, ID#0 4 사이클에 걸쳐 수행되며 각 단계별로 SRAM으로부터 지정된 주소에 해당하는 데이터를 읽어오게 된다. 이후 3 사이클에 거쳐 ALU 연산 등을 수행하는 execute 과정을 통해 출력되는 결과를 write-back#0, #1 2개의 stage를 거쳐 SRAM 쓰기 동작을 수행한다. 이 과정을 살펴보면 총 12개의 스레드 중 0번 스레드와 6번 스레드 사이에는 SRAM 사용이 중복됨이 없음을 알 수 있다.

이러한 관계는 #1과 스레드 #7을 포함한 스레드 #5와 스레드 #11 사이에도 동일하게 적용된다. 이에 기반하여 SRAM 접근이 중복되지 않는 스레드에 한해 동일한 SRAM을 사용하는 interleaved 방식으로 STBs 구성할 수 있다. 또한 이를 적용하기 위해 기존의 dual port SRAM의 구성을 single port SRAM으로 구성하여 한번에 하나의 데이터를 처리하도록 함으로써 SRAM의 크기를 줄이고 Shader의 최대 동작 속도 역시 높일 수 있었다.

그림 5.는 중복되지 않는 두 개의 STBs를 통합한 구조를 적용한 Fragment Shader의 내부 코어와 외부 fixed hardware간의 인터페이스 구조이다. 6:1 MUX 1단만을 사용하여 기존의 12:1 MUX 구조보다 muxing delay를 반으로 줄일 수 있으며 전체 SRAM의 크기를 감소시킬 수 있도록 설계하였다.

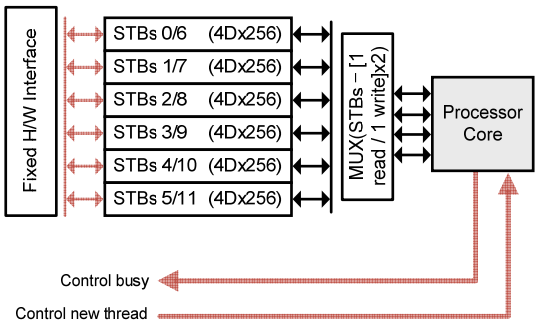

그림 5. 제안하는 Shader 메모리 입출력 구조
Fig. 5 Proposed memory structure of a Shader

또한 기존 구조에서의 STBs의 외부 입출력은 원활한 전송을 위해 프로세서 내부 버스 인터페이스를 통해서 이루어졌다. 이는 Shader가 fixed hardware의 내부 모듈로서 존재할 때에는, 각 STBs와 Shader 외부 버퍼를 직접 연결하고 간단한 컨트를 신호를 추가하는 것으로 이를 대체할 수 있다.

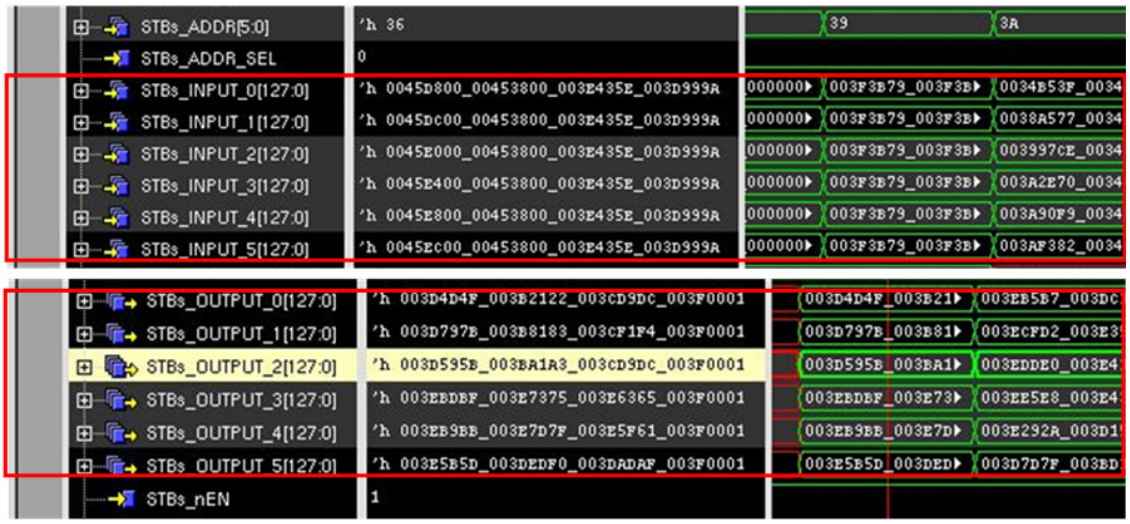

그림 6 Fragment Shader 시뮬레이션 결과
Fig. 6 Fragment Shader simulation result

본 구조에서는 프로세서 내부에 존재하는 각 6개의 STBs와 외부 포트와의 직접 연결 인터페이스를 구축하였다. 그림 6.은 개선된 구조의 Fragment 입력과 출력에 이상이 없음을 검증한 시뮬레이션 파형을 나타내고 있다.

Fragment 입력과 출력은 프로세서 코어에서의 busy 신호가 비활성화 된 시점에서 이루어지며 입력값 설정 후에 New-thread 신호로 Shader 동작을 제어할 수 있다. 이를 통해 Shader 외부 버퍼와 한 클럭에 4D x 6개의 데이터 전송이 가능하여 Shader의 유휴시간을 크게 감소시켜 성능을 높일 수 있다.

V. 검 증

동작 검증 및 코어 및 외부 인터페이스에서 STBs로 접근 시에 발생하는 muxing delay를 측정을 위해 Xilinx 사의 Virtex II-pro(xc2vp30) FPGA 보드를 사용하였으며, FPGA 검증 툴은 Xilinx ISE 8.2를 사용하였다. Fragment 입출력 성능치 측정은 Mentor Graphics사의 ModelSim 6.1 시뮬레이터를 사용하였다. 그림 7.은 기존 Fragment Shader 구조와 제안하는 구조를 비교한 표이다.

Fragment Architecture		Mux Delay (ns)	Fragment Input Counts (cycle)	Fragment Output Counts (cycle)
FPGA	12 STBs	9.3	120	36
	8 STBs	7.4	80	24
	Optimized 12 STBs	4.6	20	6

그림 7. 성능치 비교
Fig. 7 Performance comparison

본 논문에서는 제안하는 Fragment Shader의 성능 측정을 위해 기존에 설계된 버스 인터페이스 기반의 12 스레드/ 8스레드 구조의 Shader를 비교하였다. Fragment input counts와 output counts는 Shader instruction을 제외한 각 스레드당 한 개의 픽셀 데이터 처리를 위해 요구되는 Varying, coordinate 입력에 지연되는 클럭과, Fragment 연산을 마치고 Shader 외부 버퍼에 Fragment 출력을 내보내는데 소요되는 클럭 카운트이다. 각 프로세서가 최대의 효율을 낼 수 있는 보유한 스레드를 모두 동작시켰을 경우를 비교하였으며 데이터 개수와 입출력에 소요되는 시간은 그에 비례한다.

동일 시점에서 메모리 접근의 중복이 없는 STBs를 통

합한 구조에서는 MUX delay가 약 4.6ns로 기존 구조보다 약 50%의 개선된 성능을 보인다. 또한 Fragment 연산에 요구되는 입력 데이터의 전송에 있어 12 스레드 구조에서는 스레드당 최소 10 사이클을 요구하여 총 120 사이클이 지연되나 개선된 구조에서는 이를 약 78% 단축하였다. 따라서 제안하는 구조의 Fragment Shader는 명령어 수행시간을 제외하고 fixed hardware로의 12개의 픽셀 데이터를 출력하기까지 총 26 사이클만을 소모하게 되어 Shader의 활용시간을 단축하여 전체 그래픽 파이프라인의 지연을 크게 개선할 수 있다.

Ⅵ. 결 론

Programmable Shader의 설계에 있어 레지스터 그룹의 효율적인 관리는 코어의 복잡도를 줄이고 파이프라인 수행 속도를 향상시킨다. 본 논문에서는 interleaved STBs 구조를 적용하여 각 스레드의 STBs의 읽기 / 쓰기 동작 시 MUX 딜레이를 반으로 줄일 수 있었다.또한 외부 입출력에 대한 버스인터페이스를 최적화하여 dedicate hardware 와의 결합에서 발생할 수 있는 병목현상이 약 78% 개선되었다.

참고문헌

[1] Jeong-Ho Woo, et al. "A 195mW, 9.1 MVertices/s fully programmable 3D graphics processor for low power mobile device", Solid-State Circuits Conference., pp.372-375, 2007.

[2] James C. Lelterman, "Learn Vertex and Pixel Shader Programming with DirectX9", Wordware Publishing, Inc., pp.181-222, 2004.

[3] Dong Young Yeo, Woo-young Kim, Kwang-yeob Lee, Jae-chang Kwak, "Multi Port Register Architecture for Mobile Shader Processor," 대한전자공학회 SoC학술대회.May, pp. 401-404, 2009.

[4] Hyung-Ki Jeong, "A Multi-thread Processor Architecture With Dual Phase Variable-Length Instructions", ITC-CSCC., pp. 209-212, 2008.

저자소개

박태룡(Tae-ryoung Park)

1985년 한양대학교 수학과 이학사
1987년 한양대학교 수학과 이학석사
1995년 한양대학교 수학과 이학박사

1994년～현재 서경대학교 소프트웨어/컴퓨터공학과 교수

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