3D 그래픽 쉐이더 프로세서를 위한 고효율 연산기 구조
(An Architecture of a high efficient ALU for 3D Graphics Shader Processor)

[1]김우영, [1]이보행, [1]이광엽, [2]박태룡
[1]서경대학교 컴퓨터 공학과, [2]서경대학교 컴퓨터 과학과
Woo-Young Kim1, Bo-Haeng Lee1, Kwang-Yeob Lee1, Tae-ryung Park2
1Dept. of Computer Engineering Seokyeong University,
2Dept. of Computer Science Seokyeong University

요 약

최근 모바일 기기에서도 고성능 그래픽 효과가 요구되면서 다양한 연산 처리를 하는 프로그래머블 쉐이더가 필요하게 되었다. 이러한 이유로 프로그래머블 쉐이더 프로세서의 ALU는 기존에 비해 상대적으로 커지게 되었다. 이 논문에서 제안하는 듀얼 페이지 구조는 프로그래머블 쉐이더에서 상대적으로 커진 ALU 하나를 이용하여 동시에 두 개의 연산 처리를 가능하게 하는 구조이다. 이러한 구조를 사용하여 기존 구조에 비해 평균 40%의 성능을 개선 하였다

Abstract

In this paper, we propose a new programmable shader architecture based on an effective ALU operation. Today's mobile devices need the programmable shader processor for a three-dimensional(3D) graphics[1]. The programmable shader processors require a lager ALU than a fixed pipeline ALU used previously. The proposed ALU architecture is able to execute two different arithmetic operations at the same time. Two instructions which need exclusive ALU operations are inserted into instruction decoders in parallel. Experimental results show the number of instruction cycles can be substantially reduced up to 40%.

Keywords : Programmable Shader, Graphics OpenGL, Effetive ALU

Ⅰ. 서 론

최근 3D 그래픽이 다양한 사업 분야에 사용되어 지면서 사람들은 좀 더 실사에 가깝고 고성능의 그래픽을 원하게 되었다. 이러한 요구를 충족시키기 위해 그래픽 프로세서는 복잡한 연산을 필요로 하게 되었다. 기존에 사용 되어진 그래픽 프로세서는 그래픽 파이프라인 처리 과정 중에 연산량이 많은 부분만 하드웨어로 구현하고 실사에 가까운 표현을 하기위한 복잡한 연산을 하는 부분은 소프트웨어로 처리하는 Partially Programmable Pipeline 방식을 사용하였다. 최근에는 좀 더 성능을 높이기 위해 그래픽 파이프라인의 모든 처리 과정을 하드웨어로 처리하는 방법을 사용하게 되었다. 그래서 정점 쉐이더(Vertex Shader) 와 픽셀 쉐이더(Pixel Shader) 그리고 Rasterization 까지 하나의 프로세서에서 처리 할 수 있는 프로그래머블 쉐이더 프로세서 구조로 발전하였다[1].

제안하는 프로세서는 OpenGL ES 2.0 API를 지원하는 프로그래머블 프로세서를 설계하기 위해서 [그림 1]에 나타나는 연산기들이 필요하게 되었다. 많은 연산이 하나의 연산기에 구현되면서 한 번에 하나의 연산만 처리하는 기존 방식 구조는 연산기를 비효율적으로 사용 하였다. 연산기를 효율적으로 사용하고 성능을 높이기 위하여 하나의 명령어로 최대 두 개의 연산을 처리하는 듀얼 페이지 구조를 갖는다. 이 구조를 사

용하여 하나의 연산기내에 구현되어 있는 연산 중에서 서로 다른 연산을 한 스테이지 내에 처리할 수 있게 되었다. 한번에 두 개의 연산을 처리함으로서 프로세서 성능이 최대 두 배 증가되었다.

Ⅱ. 관련 연구

고정된 파이프라인의 그래픽 프로세서는 각 단계에 특화된 연산을 하기 위한 연산기만 존재한다. 정점 쉐이더 프로세서의 경우는 곱셈기와 덧셈기만으로도 충분하게 구현할 수 있다. 하지만 프로그래머블 통합쉐이더구조는 정점 처리와 픽셀 처리를 하나의 프로세서 할 수 있어야 해서 더 복잡한 연산을 필요로 하게 되었다. 하나의 연산기에 필요한 연산을 모두 포함하게 되면서 연산량이 많아진 프로그래머블 그래픽 프로세서는 연산기의 크기가 확장되는 결과를 가져오게 된다.

Instruction Type	Instruction Set	
	Normal	Alternate
Arithmetic	MOV (move)	MVS(move status registers)
	ADD (add) MUL (multiply)	
	CMP (compare)	reserved
	RCP (Reciprocal) MAN (Mantissa)	RSQ (Reciprocal square root) EXP (Exponent)
	FLR (Floor) CONV (data type conversion)	FRC (Fraction) reserved
	AND (bit logic and) XOR (bit logic exclusive or)	OR (bit logic or) SHF (logical or arithmetic shift)
Coordinate	PRED (predicate coordinate) ADDR (address coordinate)	
Control	BRC (branch) MEM (memory operation)	

그림 1. 연산기에 필요한 명령어의 셋
Fig. 1. Instruction Set of ALU

또한 API가 기존 OpenGL ES 1.1 API의 기반에서 OpenGL ES 2.0 API 기반으로 발전되면서 Looping, Dynamic-branch 과 같은 기능이 추가되었다. 모든 OpenGL ES 2.0 API 기능을 지원하고 프로그래머블 쉐이더를 구현하기 위해 [그림 1]에 나타나는 모든 명령어를 필요로 하게 되었다[2].

그래픽 프로세서가 처리하는 대부분의 스트림 데이터들은 x,y,z,w(좌표) 또는 r,g,b,a(색상) 등 4개의 컴퍼넌트 형태로 되어 있다. 그래서 대부분의 그래픽 프로세서는 효율적으로 데이터를 처리하기 위해서 [그림 2]와 같이 4개의 연산을 동시에 할 수 있는 구조로 이루고 있어 같은 연산기가 4개가 필요하다[3]. 제안하는 프로세서구조의 연산기도 총 4개로 구성되어 있기 때문에 제안하는 프로세서 연산기는 FPGA Vertex-5에서 검증하였을 때 전체 프로세서에서 60%의 크기를 사용하였다. 연산기가 프로세서에서 큰 비중을 차기하기 때문에 연산기를 새롭게 추가하지 않고 성능을 향상 시키는 방법이 필요하였다.

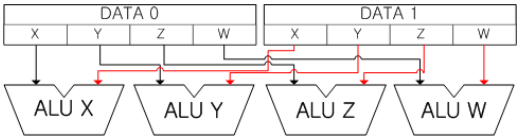

그림 2. 그래픽 프로세서의 기본 데이터 흐름 구조
Fig. 2. Basic data path for Graphics processor

Ⅲ. 쉐이더 구조

제안하는 프로그래머블 쉐이더 구조는 [그림 3]와 같다. 제안하는 프로세서의 명령어는 총 4개의 유닛 명령어로 이루어진다. 두 개의 유닛 명령어가 조합되어 하나의 Micro operation을 가지는 하나의 Phase가 되는 것이다. 그래서 VLIW(Very Long Instruction Words)[4]와 유사한 방식으로 최대 두 개의 Phase를 가지는 명령어가 처리되는 과정을 그림에서 보여준다. 하나의 명령어가 Fetch되면 한번의 ID(Insctruction Decoder) #0 단계에서 Phase #0의 명령어를 디코딩 하고 ID #1에서 Phase#1의 명령어를 디코딩한다. 각 Phase마다 ID 단계가 끝나고 Source Coordinate와Swizzle 과정을 거쳐 Execution 단계에 연산을 할 데이터가 전송 된다. 그림에서 보이듯 Phase#0이 처리하는 데이터 Souce#0 A,B 두 개와 Phase#1이 처리하는 데이터 Source#1 A,B 두 개의 데이터로 두 개의 연산이 Execute 단계에서 처리된다. 각 소스는 모든 컴퍼넌트(X,Y,Z,W)를 처리하는 데이터 구조를 가진다. 즉 4개의 컴퍼넌트에서 4개의 연산을 한 번에 하고 두 개의 Phase는 하나의 컴퍼넌트에 서로 다른 연산을 동시에 처리 하는 과정을 나타내는 것이다. 하나의 컴퍼넌트만 사용하여 각 Phase가 처리되는 과정은 다음 챕터에 자세하게 설명되어 있다. 위와 같은 과정을 통해 하나의 명령어에서 서로 다른 두 개의 연산이 처리되고 이로 인해 최대 두 배의 성능을 향상 시킬 수 있게 되었다.

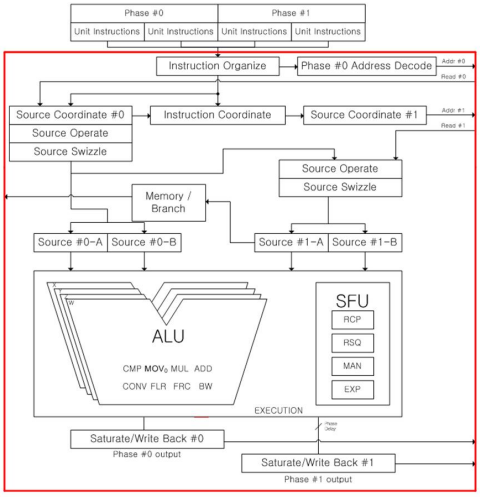

그림 3. 쉐이더 구조

Fig. 3. Shader Architecture

[그림 4]는 프로세서 처리과정중 execution 단계의 과정과 연산기를 자세하게 표현한 그림이다. execution단계 총 3단계로 나뉘어 처리된다. 제안하는 프로세서는 API를 지원하기 위해 꼭 필요하지만 자주 쓰이지는 않는 연산을 따로 SFU(Special Function Unit)에 만들어 총 프로세서에서 하나만 사용하고 공통 연산기는 각 컴퍼넌트마다 사용하여 4개를 사용한다. SFU는 프로세서에 하나만 존재하기 때문에 SFU에 구현되는 연산은 프로세서에 한번만 처리 할 수 있는 구조를 가진다.

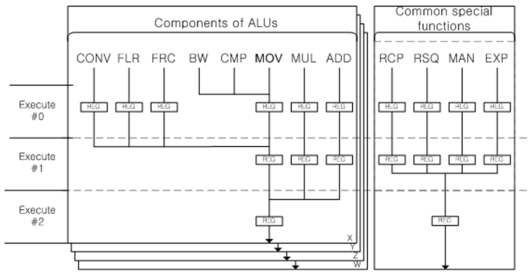

그림 4. 제안하는 연산기 구조

Fig. 4. ALU architecture of Proposed Processor

III. 듀얼 페이지 구조

하나의 컴퍼넌트에서 듀얼 페이지 구조를 사용한 간단한 데이터 패스 과정이 [그림 5] 에 나타나 있다. 앞서 프로세서 구조에서 설명하였듯이 제안하는 듀얼페이지 구조는 VLIW(Very Long Instruction Words)와 유사하게 각자 자기의 Micro operation을 가지는 Phase가 두 개로 나뉘어 처리된다. VLIW는 구조는 한 가지 이상의 연산을 하게 되면 한 개 이상의 연산기가 필요하다. 하지만 제안하는 듀얼페이지 구조는 하나의 연산기를 두 개의 Phase가 공유하기 때문에 한 개 연산기만 사용하여 두 개의 연산기가 있는 것과 같은 효과를 낼 수 있다. 예를들어 [그림 6]에 X컴퍼넌트만 사용한 간단한 연산 예시를 보면, 각 Phase에서 연산기 내에 있는 연산 중 ADD,MUL 서로 다른 두 연산을 사용 할때는 문제가 되지 않는다. 하지만 제안하는 구조는 두 개의 Phase가 공유된 하나의 연산기를 사용하기 때문에 ADD,ADD와같이 다른 Phase가 서로 같은 연산을 수행 할 수 없다.

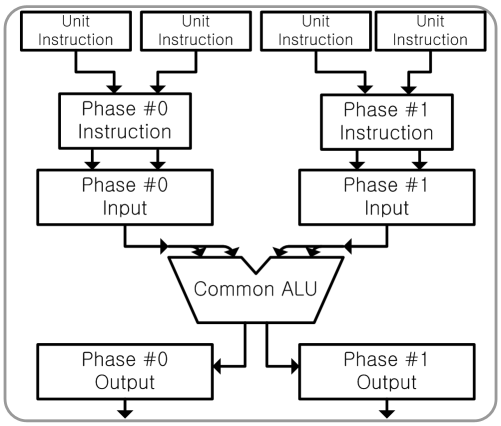

그림 5. 듀얼 페이지 데이터 패스

Fig. 5. Daul Phase Data Path

그림 6. 예시

Fig. 6. Example

IV. 검증

[그림 7]은 한 스테이지에 하나의 연산만 처리하는 기존 구조와 제안하는 구조에서 가능한 방식의 연산을 비교하여 간단한 명령어 표현으로

나타낸 예시이다. EX1)은 ADD 연산과 MUL연산을 할 때 서로 연산 스테이지 숫자를 비교한 것이다. 연산기가 하나만 있을 경우 기존 명령어는 한 파이프라인에서 한 스테이지에 하나의 연산만 수행 가능 하기 때문에 두 개의 연산을 처리를 위해서 두 개의 스테이지를 사용해야 한다. 하지만 제안하는 구조는 같은 연산이 아닌 경우 한번에 두 가지 연산을 할 수 있기 때문에 그림 오른쪽과 같이 하나의 스테이지에 두 가지 연산을 처리 하여 50%의 성능 향상을 보인다. 그래픽 프로그램에서 가장 많이 쓰이는 예시로 4x4 matrix 곱셈연산 즉 DOT 명령어를 4번 했을 때 쓰이는 명령어를 비교한 그림이 EX2) 이다. 기존의 한 스테이지에 하나의 연산만 할 때 총 10개의 스테이지로 처리 할 수 있는 연산이다. 하지만 제안하는 프로세서에서는 처리 과정 중4개의 연산이 다른 연산에 영향을 주지 않아 합쳐서 동시에 수행 할 수 있었다. 그래서 제안하는 프로세서는 4x4 Matrix 곱셈 연산의 경우 10개의 스테이지 중 4개의 스테이지를 줄여서 40%성능 향상을 보인다.

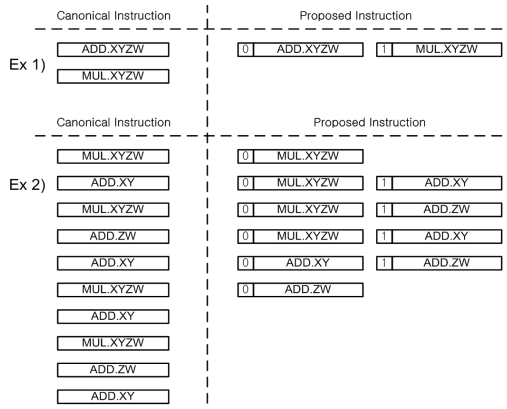

그림 7. 검증 예시

Fig. 7. Verification Examples

[그림 8]는 그래픽 프로그램에서 많이 쓰이는 기능[1][2]을 기존 구조와 비교하여 나타낸 성능 향상 결과이다. [그림 7]의 EX2)에서 보듯이 동시에 연산 수행이 불가능 한 경우도 있기 때문에 각 기능마다 성능 향상이 다르다. 3번의 DOT 연산이 필요한 3x3Matrix 곱셈 연산은 8개에서 3개의 스테이지를 줄이고 Conditional indirect branch는 3개중 하나의 스테이지를 줄였고 Nested function call and return은 6개중 4개 스테이지를 줄였다. 그래서 검증한 기능의 전체 수치 평균치를 계산해본 결과 약 40%의 연산 스테이지를 줄여 성능을 향상 시킬 수 있었다.

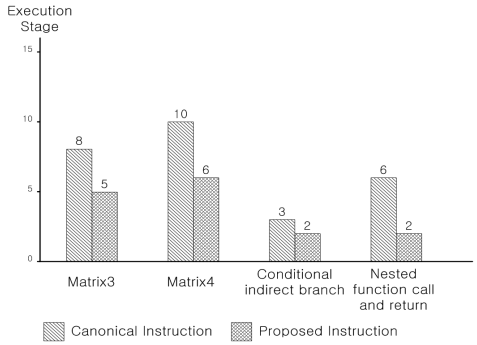

그림 8. 명령어 사이클 비교

Fig. 8. Comparison of instruction cycles

Ⅳ. 결 론

본 논문에서는 기존의 하나의 연산기에 하나의 연산만 할 수 있었던 방식과는 다르게 하나의 연산기에서 두 개의 서로 다른 연산을 처리하는 듀얼페이지 구조를 제안 하였다. 이 구조방식을 사용하여 연산기를 효율적으로 사용할 수 있었고, 그래픽 프로세서에서 많이 사용 되는 기능을 비교해 보았을 때 전체적으로 평균 40%의 명령어 cycle을 줄일 수 있었다.

*본 논문은 중소기업청 중소기업기술혁신사업과 ETRI 시스템 반도체 진흥센터 지원으로 제작되었으며 설계에는 IDEC 지원 장비를 활용하였습니다.

참 고 문 헌

[1] Kris Gray , "Microsoft DirectX 9 Programmable Graphics Pipeline" Microsoft Press

[2] OpenGL ES 2.0 specification http://www.khronos.org/opengles/

[3] Donghyun Kim, Kyusik Chung, Chang-Hyo Yu, "An SoC With 1.3 Gtexels/s 3-D Graphics Full Pipeline for consumer Applications", Solid-State circuits Conference. 2005. Digest of Technical Paper. ISSCC. 2005 IEEE International. pp. 10-10. Feb 2005.

[4] Lee Weng Fook, "VLIW Microprocessor Hardware Design" McGraw-Hill 2007.09,02