■ 2002년 정보과학 논문경진대회 수상작

3차원 그래픽 지오메트리 연산을 위한 벡터 지오메트리 엔진의 설계

(The Design of VGE(Vector Geometry Engine) for 3D Graphics Geometry Processing)

김 원 석 † 정 철 호 †† 한 탁 돈 †††
(Won-suk Kim) (Cheol-ho Jeong) (Tack-Don Han)

요 약 3차원 그래픽 가속기는 지오메트리 처리(geometry processing)와 레스터라이저(rasterizer)로 구성된다. 본 논문에서는 지오메트리 처리를 고속으로 수행할 수 있는 벡터 형태의 처리 구조(VGE)를 제안하였다. 특히 기존의 부동소수점을 계산할 수 있는 구조에 4개의 FADD, FMUL, 128개의 벡터 레지스터를 추가하여 지오메트리 연산을 가속했으며 VGE와 비슷한 H/W 비용을 갖는 Hitachi의 SH4와 비교했을 때 평균 4.7배의 성능향상을 보였다. 또한 성능 평가를 위해 범용프로세서 시뮬레이터인 Simplescalar를 수정하여 시뮬레이터를 제작했으며 Viewperf Benchmark를 입력으로 사용하였다.

키워드 : 지오메트리 처리, 시뮬레이터

Abstract 3D Graphics accelerator is usuallly composed of two parts, geometry engine and rasterizer. In this paper, VGE(Vector Geometry Engine) which exploits vertex-level parallelism is proposed. In VGE, Common Floating-Point Unit by adding four-FADD, four-FMUL unit and 128-vector register accelerates geometry calculation. In comparison with SH4, Performance result show that the VGE can achieve performance gain over 4.7 times. To evaluate VGE performance, we make simulator to rebuild Simple-Scalar, general purpose processor simulator. In simulator model, we use Viewperf-benchmark.

Key words : geometry calculation, simulator

1. 서 론

3차원 그래픽 가속기는 지오메트리 처리(geometry processing)와 레스터라이제이션 단계(rasterization)로 구성된다. 지오메트리 처리는 객체에 대한 공간상의 위치변화와 빛에 대한 계산이 이뤄지는 과정이며 좌표변환, 노말 변환, 라이팅(lighting), 클리핑(clipping), 프로제션(projection), 뷰포트(viewport)변환으로 구성된다. 레스터 라이제이션 단계는 지오메트리 처리에 의해 변환된 프리미티브(점, 선, 삼각형 등)들을 프레임 버퍼(frame buffer)에 픽셀 데이타 값으로 변환시키는 과정이다.

† 비 회 원 : 대우 일렉트로닉스 리빙연구소 리빙회로팀 연구원
wskim3@web.dwe.co.kr
†† 비 회 원 : 연세대학교 컴퓨터과학과
chjeong@kurene.yonsei.ac.kr
††† 종신회원 : 연세대학교 컴퓨터과학과 교수
hantack@kurene.yonsei.ac.kr
논문접수 : 2002년 6월 19일
심사완료 : 2003년 10월 15일

1999년대까지, PC에서 사용하는 대부분의 3D 그래픽 가속기는 레스터라이져 단계만 처리했으며 CPU가 지오메트리 처리를 수행하였다. 또한 CPU의 지오메트리 처리능력을 높이기 위해 AMD 3DNow!, Intell MMX2와 같이 수퍼스칼라 CPU의 SIMD(Single Instruction Multiple Data)연산을 지원하고 있다[1]. 하지만 이러한 방법은 지속적으로 증가하는 3D 그래픽스 어플리케이션의 정점처리 요구량을 감당하기에는 역부족이었다.

많은 양의 정점들을 처리해야 하는 어플리케이션 환경에서 CPU 처리 능력만으로 지오메트리 처리를 하기에는 한계에 도달했으며 H/W 제작 가격의 하락과 함께 3D 그래픽 가속기에 지오메트리 엔진을 장착하는 것이 필수적으로 되었다[2]. 예를 들어 Geforce 2에 장착한 nVIDIA의 T&L엔진, Radeon에 탑재한 ATI의 Charisma엔진과 같은 파이프라인 방식의 지오메트리 엔진이 그래픽 시장을 지배하였다.

그러나, 최근에는 3차원 그래픽에도 실감영상이 강조되면서 API에서도 버텍스 쉐이더(vertex shader), 픽셀

쉐이더(pixel shader)와 같은 실감영상을 위한 새로운 그래픽 알고리즘이 지원되었다[3]. 하지만 파이프라인 방식의 지오메트리 엔진으로는 이러한 알고리즘이 적용될 수 없다.

본 논문에서는 프로그램이 가능하여 어떠한 그래픽 알고리즘도 적용시킬 수 있으며 고속으로 지오메트리 과정을 처리할 수 있는 VGE(vector geometry engine)를 제안하였다. 이 구조는 기존의 부동소수점을 계산할 수 있는 구조에 4개의 FADD, FMUL, 128개의 벡터레지스터를 이용했으며, 추가된 H/W를 이용하여 벡터 연산을 수행해 지오메트리 연산을 효율적으로 가속할 수 있다. 또한 성능평가를 위해 제안하는 구조와 비슷한 H/W 비용을 갖는 Hitachi의 SH4를 비교 대상으로 하여 시뮬레이션을 수행하였다. 시뮬레이터는 수행구동(execution driven) 시뮬레이터인 Simplescalar의 소스코드를 수정하여 제작했으며, VGE와 SH4의 어셈블리 명령어를 입력으로 파이프라인으로 처리된다. 또한 분석적인 모델을 통해 최적의 성능을 나타내는 VGE의 레지스터 개수와 데이타 메모리 엔트리를 결정하였다.

본 논문의 구성은 다음과 같다. 2절에서는 관련연구를 알아보고, 3절에서는 제안하는 VGE 구조와 특징을 살펴본다. 4절에서는 시뮬레이션 환경을 설명하며, 5절에서는 시뮬레이션의 결과 및 평과를 알아보며, 6장에서는 결론을 논한다.

2. 관련연구

이 장에서는 시뮬레이션을 통한 지오메트리 처리과정의 특징과 현재까지 지오메트리 연산의 가속을 위해 제안되었던 H/W 구조를 알아보고자 한다. 특히 시뮬레이션은 범용프로세서인 MIPS ISA-IV를 모델로 하여 Viewperf 벤치마크를 실행시켰다.

2.1 지오메트리 처리과정의 특징

3D 그래픽스의 지오메트리 처리는 데이타의 특성상 부동 소수점 연산이 대부분이다. 그렇기 때문에 지연시간과 계산비용이 상당히 크다. 또한 지오메트리 계산은 좌표를 대상으로 처리하기 때문에 데이타들간의 병렬성은 상당히 크다. 특히 하나의 좌표 데이타를 처리하는 연산들의 대부분은 좌표계 변환으로 행렬 곱셈(matrix multiplication)이 대부분이다. 이때 사용되는 부동 소수점 곱셈과 부동 소수점 덧셈 명령어들은 서로 독립적으로 수행될 수 있다.

시뮬레이터로 범용 프로세서인 MIPS ISA-IV의 어셈블리 명령어로 Viewperf 벤치마크를 실행시켰을 경우 각 지오메트리 단계의 실행시간의 분포를 조사하여 다음과 같은 결과 그림 1을 보였다.

그림 1에서 라이팅계산은 Awadvs-04(35%), ProCDRS-03(29%), DRV-07(18%), DX-06(17%)의 분포를 나타냈다. 참고로 LIGHT-04는 라이팅 계산을 하지 않는다. 나머지는 대부분 변환연산으로 Awadvs-04(61%), ProCDRS-03(77%), DRV-07(78%), DX-06(79%), LIGHT-04(94%)였다.

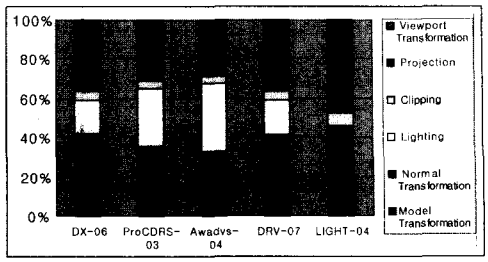

그림 1 MIPS ISA-IV를 모델로 Viewperf 벤치마크를 시뮬레이터의 데이타로 실행시켰을 때 각 지오메트리 단계의 비율

평균 76%가 변환(Model, Normal, Viewport, Projection)연산이고 20%가 라이팅 연산이었다. 변환단계는 행렬 곱셈이 대부분이기 때문에 지오메트리 프로세서의 성능을 개선시키기 위해서는 행렬 곱셈과 라이팅 계산을 빨리 처리해야 한다.

2.2 지오메트리 가속을 위한 H/W 구조

지오메트리 처리를 가속시키는 방법은 부동 소수점 연산 처리 방법과 모델 데이타의 처리 방법에 따라 여러 가지로 나눌 수 있다. 좌표 데이타 병렬성을 이용하여 처리하는 SIMD(Single-Instruction stream Multiple-Data stream), 데이타와 명령어의 병렬성을 이용하는 MIMD (Multiple-Instruction stream Multiple-Data stream), SIMD와 MIMD방식을 적절히 혼용한 하이브리드(hybrid) 방식 등으로 나눌 수 있다. 또한 지오메트리 처리 단계들이 순차적으로 수행되는 것을 이용하여, 이들 단계들을 파이프라인으로 구성하여 처리하는 파이프라인 방식이 있다.

파이프라인 구조는 지오메트리 단계마다 기능 유닛을 두어 파이프라인을 형성시켜 가속시키는 방법이다[4,5]. 또한 SIMD, MIMD, 하이브리드 구조는 3D 그래픽 데이타와 명령어의 병렬성을 이용하여 다수의 기능유닛이 같은 동작을 하여 처리율(throughput) 을 높이는 병렬 구조이다[6].

SIMD, MIMD, 하이브리드 구조는 동시에 여러 개의 3D 그래픽 데이타가 처리되는 공통점이 있고 파이프라인 구조와는 확연히 구별되므로 본 논문에서는 이러한 구조를 병렬 구조라 명하며 파이프라인 구조와 비교하며 기술하겠다.

2.3 파이프라인 구조

파이프라인구조는 지오메트리 처리 단계의 일부분이

나 전체를 하나의 파이프라인 단계로 정의하고, 각 단계에 맞는 기능유닛들을 둠으로써 파이프라인을 형성시켜 지오메트리 처리를 가속시키는 방법이다. 이 구조는 단순히 각 단계의 오퍼레이션에 따라 기능유닛을 구성한 것이기 때문에 연산기와 레지스터 이외의 부가적인 유닛(sequencer, macro-code store)이 필요 없다. 또한 각 기능유닛에 데이타를 분배하는 과정이 필요 없으며, 프로세서의 제어가 다른 구조에 비해 훨씬 간단하다. 반면에 시간이 가장 오래 걸리는 단계에 의해 전체 성능이 좌우되며, 각 단계들의 연산의 양이 균형을 유지하지 못하기 때문에 전체의 성능의 제한이 있게 된다. 또한 새로운 알고리즘을 위한 하드웨어 수정이 어려워 확장성에 문제가 있다[7].

2.4 병렬 구조

벡터 구조는 좌표 데이타의 병렬성을 이용하여 처리하는 방법이다. 지오메트리 과정의 계산은 공간상의 좌표 단위로 이루어 진다. 좌표는 4개의 성분(x, y, z, w)으로 이루어 진다. 이러한 좌표의 성분들은 서로 독립적으로 수행 될 수 있는데 이러한 좌표 성분의 병렬성을 이용하여 처리율을 높이는 방법이다. 전체의 기능유닛이 같은 동작을 하기 때문에 구현이 복잡하지 않으며, 부가적인 장치들을 각 기능유닛이 공유할 수 있다. 또한 그래픽 데이타의 특성상, 풍부한 병렬성 때문에 기능유닛을 더 추가할수록 선형적인 성능향상을 기대할 수 있다.

벡터 구조로서 가장 대표적인 지오메트리 엔진은 Hidachi의 SH4와 Sony의 이모션 엔진(emotion engine)이다. SH4는 게임기 드림캐스트(DreamCast)에 탑재되어 있으며 주로 지오메트리 연산을 한다. SH4는 32개의 부동 소수점 레지스터를 갖고 있으며 4개의 부동소수점 레지스터가 한 개의 벡터 레지스터로 리맵핑(remapping)되어 벡터연산의 연산자로 활용된다. 또한 Inner Product H/W를 갖고 있어 다음의 식 (1)의 연산을 한 개의 명령어로 처리할 수 있다[8].

$$m0*ox + m4*oy + m8*oz + m12*ow \quad (1)$$

PlayStation 2에 탑재되어 있는 이모션 엔진은 동시에 64비트 명령어 2개가 수행되고 한 개의 64비트 명령어는 2개의 32비트 명령어를 포함하는 VLIW 형태이다. 또한 32개의 128비트 부동소수점 레지스터를 가지며 한 개의 레지스터는 32비트 부동소수점 값 4개를 갖는다[9].

이모션 엔진은 2개의 VPU(Vector Processor Unit)를 갖는다. 각각의 VPU는 64비트 VLIW 명령어를 처리하며 4개의 FMAC과 Load/Store Unit, FDIV, SQRT, int ALU를 갖는다. VLIW 명령어는 32비트 Upper 명령어와 Lower 명령어로 나누어 지는데 Upper 명령어에서는 4개의 FMAC unit을 이용한 벡터연산이 이루어지고 Lower 명령어에서는 그 밖의 연산이 이루어 진다.

또한 이모션 엔진은 물체(object) 단위로 병렬적으로 처리된다. 예를 들어 게임 데이타를 처리할 때 한 개의 VPU는 배경 등의 정적인 물체의 지오메트리 계산을 하고 다른 VPU는 캐릭터 같이 움직이는 물체를 계산한다.

2.5 파이프라인 구조와 병렬 구조의 비교

지금까지 두 구조에 대해 설명한 것을 정리하면 표 1과 같다. 일반적으로 파이프라인 구조는 각 지오메트리 단계에 기능유닛을 두어 H/W 효율(optimization)을 우선시 했다. 또한 파이프라인으로 처리되어 가격당 성능비(cost-performance)가 병렬 구조에 비해서 좋다. 하지만 가장 오래 걸리는 단계에 의해 성능이 좌우되며 그 단계가 병목이 될 수 있다. 또한 지오메트리 단계마다 기능유닛을 구성하였기 때문에 H/W 구성하기에 비용이 많이 들고 새로운 그래픽 알고리즘을 적용하기 힘들다.

또한 병렬 구조는 파이프라인 구조와 비교하여 기능유닛을 추가하기가 수월하다. 즉 확장성(scalability)이 좋다. 그러나 파이프라인 구조는 모든 단계에 기능유닛을 추가해야 성능의 향상을 기대할 수 있으므로 확장성에 대한 비용이 크다.

표 1 병렬 구조와 파이프라인 구조의 비교

파이프라인 구조	병렬 구조
Fixed Processing	Flexible Processing
Limited Adaptation of new Algorithms	Unlimited Adaptation of new Algorithms
H/W Optimization	S/W Optimization
Complex for scalability	Simple for scalability
Low Resource utilization	High Resource utilization
Not utilize memory access	Utilize memory access

3. 제안하는 VGE 구조

이 장에서는 제안하는 VGE 구조의 H/W 구성과 특징을 살펴보겠다. 또한 VGE 구조에서 지오메트리 연산을 좌표단위로 처리하는 방법과 VGE 벡터 명령어에 대해서 알아보겠다.

3.1 병렬구조를 활용한 VGE의 H/W 특징

지오메트리 연산에 있어서 대부분을 차지하는 것은 변환연산과 라이팅 연산이다. 그런데 변환연산은 (x, y, z, w)의 성분을 갖는 정점단위로 이루어지고 라이팅 계산은 색의 성분인 RGB단위로 각각 이루어진다. 특히 이 연산들은 정점과 색의 곱셈과 덧셈이 대부분이기 때문에 곱셈과 덧셈을 벡터로 처리했을 때 효율적이다.

그래서 VGE는 정점을 구성하는 네 개의 성분(x, y,

z, w)과 색을 구성하는 RGB를 벡터 단위로 처리할 수 있도록 네 개의 부동소수점을 저장하는 128비트 벡터 레지스터를 지원하고, 데이타 버스 폭 또한 128비트로 하였다. 또한 곱셈과 덧셈에 벡터연산을 지원하기 위해 네 개의 FALU와 FMUL을 갖는다. 그리고 덧셈과 곱셈에 있어서 유연한 벡터연산을 위해 벡터형(vector type) 연산과 브로드 캐스트형(broadcast type) 연산을 지원한다.

특히 변환연산에는 로드/스토어(Load/Store) 명령어가 정점단위로 이루어지고 라이팅 연산에는 RGB성분을 구하는 연산이 색단위로 이루어진다. 그래서 범용프로세서로 이러한 연산을 처리할 경우 RGB성분을 각각을 로드(load)해야 한다. 하지만 VGE에서는 벡터레지스터 단위로 로드/스토어가 가능하므로 한번의 로드/스토어 명령어로 RGB 성분 모두를 처리될 수 있다.

또한 변환연산과 라이팅 연산은 모든 정점에 대해 동일한 연산을 수행하므로 명령어들의 루프(loop)가 자주 발생한다. 그래서 VGE는 128개의 벡터 레지스터를 이용하여 루프를 최대한 언롤(unroll)할 수 있도록 하여 분기(branch)로 인한 지연시간을 최소화하였다.

VGE의 H/W구성(표 2)과 블록도(그림 2)는 다음과 같다. 또한 표 3은 VGE에서 지원하는 명령어들의 지연시간을 보여주고 있다. 특히 FADD, FMUL, FCMP을 명령어를 지원하여 벡터 단위로 연산이 가능하도록 했다.

표 2 VGE H/W 구성

기능 유닛	4개의 FADD, 4개의 FMUL Integer ALU, Load/Store Unit FDIV/SQRT
레지스터	128개의 128비트 벡터 레지스터 16개의 32비트 레지스터
메모리	명령어 메모리(4KB) 데이타 메모리(8KB)
버스폭	128 비트

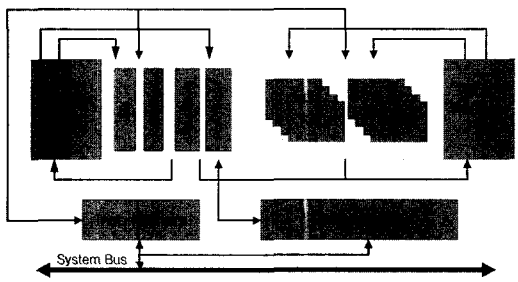

그림 2 VGE의 Block Diagram

표 3 VGE H/W 구성

Instruction	VGE
FADD/FSUB	3 cycle
FMUL	2 cycle
ROUND	3 cycle
CONVERT	3 cycle
FCMP	1 cycle
Etc(ABS,NEG)	1 cycle
FMAC	5 cycle
FDIV/SQRT	7 cycle

3.2 VGE가 지원 하는 그래픽 가속 연산

지오메트리 연산 중 95%이상을 차지하는 좌표 변환과 라이팅 연산의 가속방법을 알아보겠다.

3.2.1 VGE의 변환연산

그림 3에서 보는 것 처럼 행렬 변환의 식을 범용 프로세서로 처리하기 위해서는 16번의 곱셈과 12번의 덧셈이 필요하다[10]. 하지만 VGE는 [m0, m1, m2, m3] 벡터와 x 스칼라 형태의 벡터 곱셈이 네 번 실행되며 이러한 곱셈의 결과를 더하는 벡터 덧셈이 세 번 실행된다.

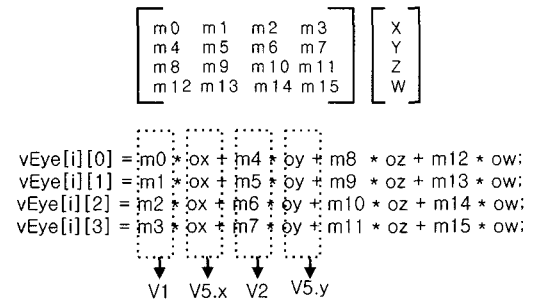

그림 3 한 개의 좌표를 변환하는 VGE의 벡터-스칼라방식

```
LV      V5                  /* 처리해야 할 좌표 => V5 Load */
MULV, V10, V1, V5.x          /* V5.x * V1 */
MULV, V11, V2, V5.y          /* V5.y * V2 */
MULV, V12, V3, V5.z          /* V5.z * V3 */
MULV, V13, V4, V5.w          /* V5.w * V4 */
ADDV, V14, V10, V11          /* V5.x * V1 + V5.y * V2 */
ADDV, V15, V12, V13          /* V5.z * V3 + V5.w * V4 */
ADDV, V16, V14, V15  /* V5.x*V1+V5.y*V2+V5.z*V3+V5.w*V4 */
SV      V16                  /* 결과를 Store */
```

그림 4 한 개의 좌표를 변환하는 데 필요한 VGE

3.2.2 VGE의 라이팅 연산

OpenGL에서 한 좌표의 라이팅계산은 식 (2)와 같이 계산한다[11].

$$VertexColor = emission_{material} +$$
$$ambient_{light\,model} * ambient_{material} +$$
$$\sum_{i=0}^{n-1}\left(\frac{1}{k_c + k_l d + k_q d^2}\right)_i * (spotlight\,effect)_i *$$
$$[ambient_{light} * ambient_{material} +$$
$$(\max\{L \bullet n, 0\}) * diffuse_{light} * diffuse_{material} +$$
$$(\max\{S \bullet n, 0\})^{shininess} * specular_{light} * specular_{material}]_i$$

식 (2) 한 개의 좌표를 라이팅계산하는 데 필요한 연산

각 용어에 대한 내용은 다음과 같다

Ambient 항:

조명의 Ambient Color값과 물체의 Ambient 특성을 곱하여 계산한다.

$$ambient_{light} * ambient_{material}$$

Diffuse 항:

조명의 Diffuse Color 값과 물체의 Diffuse 특성, 그리고 조명위치의 단위벡터 L(그림 10)과 좌표의 노말 단위 벡터의 내적의 곱으로 계산한다.

$$(\max\{L \bullet n, 0\}) * diffuse_{light} * diffuse_{material}$$

Specular 항:

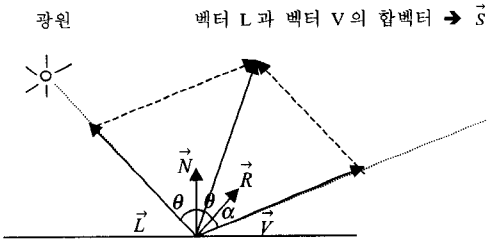

그림 5 라이팅 계산에 사용되는 벡터

L은 광원벡터, N은 좌표의 노말 벡터, R은 L의 반사 벡터이다. Specular 항은 Specular color 값과 물체의 Specular 특성, $(\vec{R}\cdot\vec{V})^{shininess}$ (그림 10)을 구하여 곱으로 계산한다. 하지만 OpenGL에서는 계산과정을 쉽게 하기 위해 $(\vec{R}\cdot\vec{V})^{shininess}$ 대신 $(\vec{S}\cdot\vec{N})^{shininess}$를 구하여 계산한다.

위의 식 계산을 위하여 OpenGL에서는 벡터 $\vec{L}$과 벡터 $\vec{V}$의 합의 벡터 $\vec{S}$를 구한 후 $\vec{S}$를 단위 벡터로 만든다. 그리고 $\vec{S}$와 $\vec{N}$의 내적을 구한다. 이때 $\vec{S}$와 $\vec{N}$의 사이의 각은 $\alpha/2$이다.

$$(\max\{S \bullet n, 0\})^{shininess} * specular_{light} * specular_{material}$$

Attenuation term

광원과 물체와의 거리를 고려한 라이팅계산을 위해 Attenuation 항을 사용한다.

d = 광원의 위치와 물체와의 거리

kc = GL_CONSTANT_ATTENUATION

kl = GL_LINEAR_ATTENUATION

kq = GL_QUADRATIC_ATTENUATION

라이팅 계산을 위해 위와 같이 곱셈과 덧셈이 많은 연산을 RGB 값에 대하여 연산을 세 번 반복해야 한다. 하지만 RGB를 하나의 벡터로 처리한다면 곱셈과 덧셈에 대해서는 한 번에 계산할 수가 있다. (덧셈과 곱셈 연산에 대해서 세 배의 성능 향상). 이 경우는 라이팅 연산에 있어서 데이타 병렬성(RGB의 세 개의 데이타)을 이용하여 가속하였다.

3.2.3 루프 언롤링 기법

루프를 구성하는 명령어 중 실제 연산이 이뤄지는 부분을 루프의 몸체라 하고 분기문의 데이타 의존성으로 인한 지연되는 부분을 오버헤드 명령어이라 한다. 루프의 몸체 부분을 반복하여 오버헤드 명령어 부분을 줄여나가는 기술을 루프 언롤링이라 한다[12].

지오메트리 연산 중 많은 비중을 차지하는 좌표 변환, 라이팅연산은 모든 좌표에 대하여 같은 연산을 반복한다. 실제로 모든 좌표에 대해서 각각 분기 명령어를 사용하여 처리한다면 많은 분기문으로 인한 오버헤드 명령어를 감수해야 한다. 이때 루프 언롤링 기술을 사용한다면 오버헤드 명령어를 줄일 수 있다. 하지만 루프의 반복되는 부분은 레지스터를 다르게 사용해야 하므로 많은 레지스터를 갖고 있어야 한번에 많은 루프의 몸체를 반복해 실행할 수 있다.

VGE구조는 128개의 많은 레지스터를 이용하여 루프 횟수를 변환연산의 경우 1/15까지 줄일 수 있다. 변환할 행렬을 로드 하는데 4개의 레지스터가 쓰이고 또한 15개의 변환할 좌표를 스토어 해야 19개의 명령어가 필요하다. 또한 한 좌표를 행렬 변환하는데 7개의 레지스터가 필요하므로 15*7+19 = 124 개의 레지스터가 필요하다. 그러므로 128개의 레지스터를 갖으면 변환 연산시 15번 루프를 반복할 수 있다.

다음 그림 6, 7은 좌표변환연산 할 때 루프 언롤링을 사용하여 오버헤드 명령어를 줄인 예이다. 그림 6의 경우 [LV V5]연산이 완전히 실행되어야 다음 명령어 [MULV, V10, V1, V5.x]가 실행될 수 있다. 이럴 때 LV 명령어 지연시간 만큼 사이클이 낭비된다. 이것을 줄이기 위해서 그림 12와 같이 [LV, V5]와 [MULV, V10, V1, V5.x]사이에 [LV, V6]이라는 V5와 의존성이 없는 명령어를 삽입하여 낭비된 시간에 다른 명령어를 처리하여 효율을 높일 수 있다. 또한 루프의 횟수도 그림 7에 비하여 절반으로 줄어들어 분기문으로 인한 오버헤드 명령어를 줄일 수 있다.

```
LV      V5              /* 처리해야 할 좌표 => V5 Load */
MULV, V10, V1, V5.x      /* V5.x * V1 */
MULV, V11, V2, V5.y      /* V5.y * V2 */
MULV, V12, V3, V5.z      /* V5.z * V3 */
MULV, V13, V4, V5.w      /* V5.w * V4 */
ADDV,  V14, V10, V11     /* V5.x * V1 + V5.y * V2 */
ADDV,  V15, V12, V13     /* V5.z * V3 + V5.w * V4 */
ADDV, V16, V14, V15  /* V5.x*V1+V5.y*V2+V5.z*V3+V5.w*V4  */
SV      V16              /* 결과를 Store  */
```

그림 6 한 개의 좌표를 변환하는 데 필요한 명령어

```
LV      V5              /* 처리해야 할 좌표 => V5 Load */
LV      V6              /* 처리해야 할 좌표 => V6 Load */
MULV, V10, V1, V5.x      /* V5.x * V1 */
MULV, V11, V2, V5.y      /* V5.y * V2 */
MULV, V12, V3, V5.z      /* V5.z * V3 */
MULV, V13, V4, V5.w      /* V5.w * V4 */
MULV,  V14, V1, V6.x     /* V6.x * V1 */
MULV,  V15, V2, V6.y     /* V6.y * V2 */
MULV,  V16, V3, V6.z     /* V6.z * V3 */
MULV,  V17, V4, V6.w     /* V6.w * V4 */
ADDV,  V18, V10, V11     /* V5.x * V1 + V5.y * V2 */
ADDV,  V19, V12, V13     /* V5.z * V3 + V5.w * V4 */
ADDV,  V21, V14, V15     /* V6.x * V1 + V6.y * V2 */
ADDV,  V22, V16, V17     /* V6.z * V3 + V6.w * V4 */
ADDV,  V20, V18, V19  /* V5.x*V1+V5.y*V2+V5.z*V3+V5.w*V4  */
ADDV,  V23, V21, V22  /* V6.x*V1+V6.y*V2+V6.z*V3+V6.w*V4  */
SV      V20              /* 결과를 Store  */
SV      V23              /* 결과를 Store  */
```

그림 7 루프 언롤링 사용해서 루프를 두 번 반복한 결과

이 경우는 루프를 두 번 반복하였지만 15번 반복한다면 모든 명령어들은 지연시간 없이 1 사이클에 1개의 명령어가 처리되어 완벽히 파이프라인으로 처리될 수 있다.

4. 시뮬레이션

이 장에서는 지오메트리 시뮬레이터 실험 환경의 구성 및 실험 과정을 살펴본다. 또한 3차원 그래픽 파이프라인의 기본적인 단계를 수행하며 단계별 결과를 보여주는 실험 환경을 설명한다. 그리고 시뮬레이션의 결과와 보여주고 평가를 제시한다.

4.1 실험 환경

실험 환경은 대표적인 그래픽 라이브러리인 OpenGL의 기능을 구현한 메사(mesa) 라이브러리의 공개된 코드를 이용하였다. 또한 수행사이클을 계산하기 위해서 지오메트리 연산을 수행하는 부동소수점 연산에 관련된 코드마다 대상 구조의 어셈블리 명령어를 삽입하였다. 그리고 메사 라이브러리를 호출할 때마다 시뮬레이터도 같이 호출하는 방식으로 수행사이클 수를 계산하였다.

시뮬레이터는 범용프로세서 시뮬레이터인 Simple-scalar의 소스코드를 수정하여 제작했으며, 어셈블리 명령어를 입력으로 파이프라인으로 처리된다. 또한 데이타 의존성과 어셈블리 명령어의 처리율과 지연시간을 고려하여 수행 사이클과 프로파일링 된 정보를 반환한다.

비교 대상은 제안하는 VGE 구조와 비슷한 H/W 비용을 갖는 SH4를 채택하였다. 또한 시뮬레이터 입력으로 SPEC View perf의 벤치 마크(Awadvs-04, DRV-07, DX-06, Light-04, ProCDRs-03)들을 사용하였다[13].

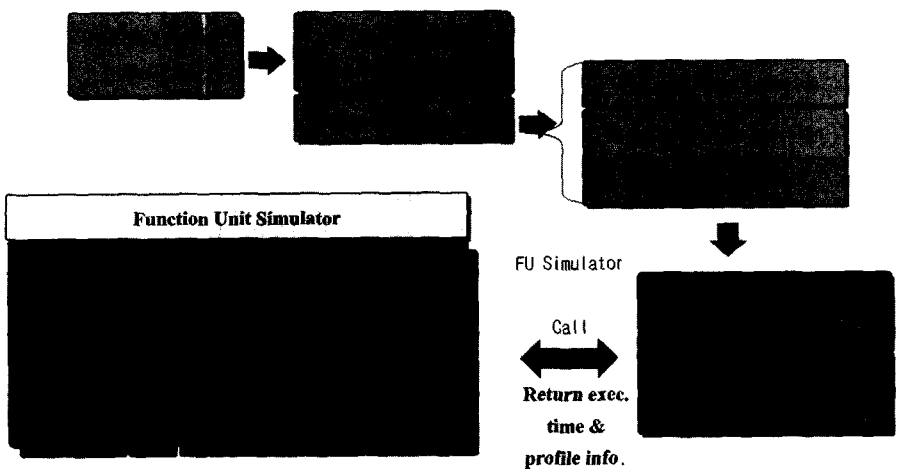

그림 8 전체적인 시뮬레이터 실험환경

4.2 실험결과 및 분석

그림 11에서 보는 것과 같이 라이팅 계산이 비교적 적은 DRV, DX, LIGHT, ProCDRs에서는 평균 네 배 이상의 성능 향상이 있었다. 그리고 라이팅 계산이 많은 Awadvs에서는 약 세 배정도의 성능 향상이 있었다.

SH4의 경우는 4개의 FMUL과 FADD를 결합하여 Inner Product 연산을 가속한 반면, VGE는 FMUL과 FADD를 분리시켜 벡터의 곱셈과 덧셈 연산에서도 가속할 수 있도록 하였다. 그래서 SH4의 경우 Dot Product의 결과가 일반적인 부동소수점 레지스터에 저장되기 때문에 vEye의 성분마다 스토어연산이 필요하다(그림 9). 반면에 벡터 스칼라 방식은 VGE 구조가 지원하는데 연산 결과가 vEye 벡터에 저장되기 때문에 한번의 벡터 스토어 연산으로 결과를 메모리에 저장할 수 있다(그림 10).

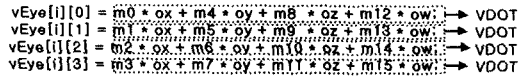

그림 9 Dot Product를 이용한 행렬연산

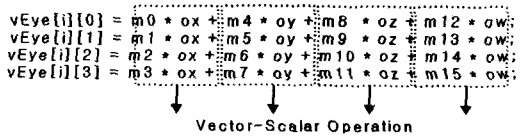

그림 10 Vector-Scalar 연산을 이용한 행렬 연산

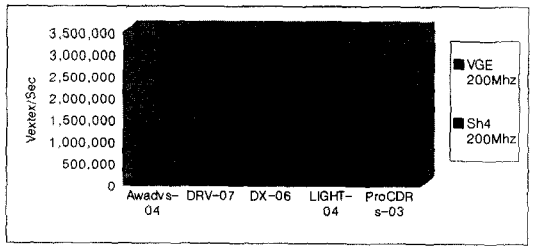

그림 11 Viewperf 벤치마크에서 SH4와 VGE의 Vertices/Sec 비교(200 MHz)

또한 SH4는 벡터레지스터가 8개이므로 루프 언롤링을 사용할 수 없었다. 하지만 VGE는 128개의 충분한 벡터 레지스터를 이용하여 루프 횟수를 변환연산의 경우 최대 15배 줄일 수 있었다. 즉, 변환할 행렬을 로드하는데 4개가 쓰이고 변환할 정점을 로드하는데 15개가 쓰인다. 그리고 한 정점을 행렬변환하는데 7개의 임시레지스터가 필요하므로 최소 124(=15*7+19)개의 레지스터가 필요하였다.

또한 벡터 단위의 스토어/스토어 연산은 데이타의 입출력이 자주 발생하는 변환연산에서 SH4보다 더 많은 성능향상이 있었다. 라이팅 연산에도 SH4의 Inner-Product 연산보다는 벡터단위의 덧셈과 곱셈이 빈번하므로 VGE가 더 효율적이었다.

4.3 메모리 크기변화에 따른 면적증가와 SH4와 면적 비교

지오메트리 엔진의 비용은 H/W 점유 면적에 비례한다. 그런데 VGE는 다른 지오메트리 엔진에 비해서 많은 벡터레지스터를 갖는다. 그래서 메모리 면적만을 놓고 보았을 때 VGE와 비슷한 H/W구성을 갖는 SH4보다 면적이 더 필요하다(표 4).

표 4 VGE, SH4의 메모리, 레지스터 구성의 따른 크기

		Width(um)	Height(um)
VGE	Instruction Memory(4KB)	985.54	375.44
	Data Memory(8KB)	1127.29	401.16
	Register(128x128bit)	2389.68	504.38
SH4	Instruction Memory(8KB)	1127.29	401.16
	Data Memory(16KB)	1525.03	508.64
	Register(32x32bit)	289.55	1021.56

0.18um 삼성 표준 셀(cell) 라이브러리를 참고하여 VGE와 SH4의 레지스터와 메모리 크기를 추정하였다(표 4). 표 4에서 보는 것과 같이 레지스터와 메모리를 합한 면적이 약 30% 증가한다.

4.4 VGE의 메모리 계층구조

실제 메인 메모리에 저장되어 있는 좌표 데이타를 로드/스토어 하기 위해서는 먼저 데이타 메모리로 좌표 데이타가 이동하고, 데이타 메모리에서 레지스터로 옮겨져야 한다. 그 후에 기능 유닛에 의해 실제 연산이 이뤄진다.

그래서 실제 지오메트리 프로세서의 처리시간은 계산시간 뿐만 아니라 메인 메모리에서 데이타 메모리로 처리해야 할 데이타를 가져오고 데이타 메모리에서 레지스터로 이동하는 시간까지 포함한다. 그래서 메모리 구조를 어떻게 결정하느냐가 성능에 주요한 요인으로 작용한다. 레지스터의 크기가 128비트라고 가정했을 때, 즉 한 개의 레지스터에 한 개의 좌표가 저장될 수 있다. n개의 좌표에 대해서 변환연산 하는데 걸리는 실제 수행시간은 식 (3)과 같이 예측할 수 있다.

식 (3)에서 R은 레지스터 중 입출력에 사용되는 레지스터의 개수이다. 보통 변환 연산에는 전체 레지스터 개수 중 1/3정도가 입출력에 사용된다. D는 데이타 메모리 엔트리의 개수, M은 메모리를 참조하는 횟수이다. 변환 연산되는 좌표는 1번의 로드와 스토어가 필요하므

로 M은 2n이다. 필요한 D.Macc_time는 데이타 메모리를 접근하는 시간이고 M.Macc_time은 메인 메모리를 접근하는 시간이다.

For n vertices,

$$Calc.time = \frac{M}{R}(4 \times R + 5) \times Clock\ cycle\ time$$

$$Exec.time = Calc.time + (M \times D.Macc\_time) + (\frac{M}{D} \times M.Macc\_time)$$

,where # of Input / output · register Size R, # of data memory entry D, # of memory reference M ($\approx 2n$)

식 (3) 지오메트리 Processor가 n개의 좌표를 변환 연산하는데 걸리는 시간

Calc.time은 n개의 좌표를 R개의 레지스터로 계산할 때 걸리는 시간을 나타낸 것이다. VGE는 네 개의 FMUL과 FADD를 갖고 있으므로 완벽히 파이프라인으로 동작한다고 가정했을 때 네 사이클이면 한 개의 좌표가 변환된다. 또한 파이프라인을 초기화하는데 5 사이클이 걸린다. 그리고 입출력 레지스터 개수만큼 한꺼번에 데이타를 가져오므로 데이타 메모리에서 데이타를 레지스터로 로드/스토어 하는 지연시간은 한 사이클로 가정한다.

또한 데이타 메모리를 접근하는 시간은 n 개의 좌표를 로드/스토어해야 하므로 2n*D.Macc_time이다. 그리고 메인 메모리를 접근하는 시간은 메모리 엔트리만큼 한꺼번에 데이타를 가져오므로 2n*M.Macc_time/D이다.

레지스터 Size와 데이타 메모리 엔트리를 무한정 늘리면 전체 실행시간이 줄어들 것 같지만 D.Macc_time와 M.Macc_time도 함께 증가하므로 그렇지 않다. 그러므로 이러한 요소들간의 Trade off점을 찾아내어 가장 최적의 수치를 결정하는 것이 중요하다.

레지스터개수와 메모리 엔트리를 결정하기 위해서 0.18um 삼성 표준 셀 라이브러리를 참고하여 실제 계산 시간을 구했다. 예를 들어, 128 개의 레지스터를 갖고 512 엔트리의 데이타 메모리를 갖는다고 가정했을 때 실제 실행시간은 다음과 같다. 계산을 위해서 좌표개수는 백만개, 클럭 사이클 타임(clock cycle time)은 500MHz라 가정했을 때 수행시간을 예측하면 다음과 같다.

$$Calc.time = \frac{2 \times 10^6}{40} \times (4 \times 40 + 5) \times 2 \times 10^{-9}$$

$$Exec.time = Calc.time + (2 \times 10^6 \times 1.80 \times 10^{-9}) + (\frac{2 \times 10^6}{512} \times 2.11 \times 10^{-9}) = 20500us$$

또한 그림 12처럼 레지스터 크기는 32개에서부터 512개까지 변화시키고 메모리 엔트리는 256개에서 32768개까지 변화시켜 실제 실행시간을 예측하였다.

그림 12에서 나타나듯이 가장 아래 쪽에 있는 곡선이 데이타 메모리가 512 엔트리인 경우이다. 또한 레지스터

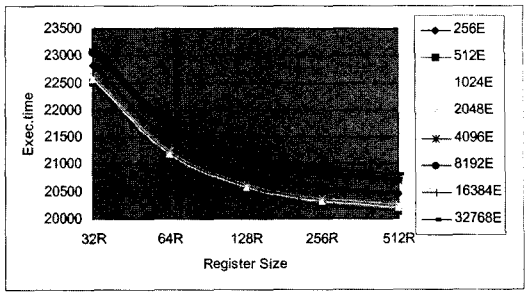

그림 12 레지스터 개수와 데이타 메모리 Size에 따른 실행시간의 변화

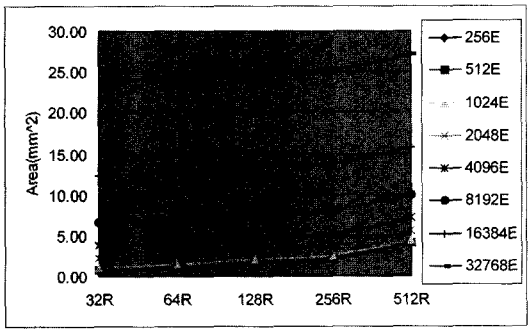

그림 13 레지스터 개수와 데이타 메모리 엔트리에 따른 면적의 변화

개수가 늘어날수록 좋은 성능을 나타내고 있다. 결국 데이타 메모리가 512 엔트리인 경우와 레지스터 개수가 512개인 경우가 가장 성능이 좋다. 레지스터 개수가 늘어나면 한 번에 많은 데이타들을 스토어 할 수 있으므로 식 (3)에서 나타나듯이 calc.time을 많이 줄일 수 있다. 그리고 데이타 메모리 엔트리가 너무 커지면 메인 메모리에서 데이타 메모리로 데이타를 접근하는 시간이 증가하므로 성능은 오히려 떨어진다.

앞에서 언급했듯이 데이타 메모리가 512 엔트리, 레지스터 개수가 512인 경우가 가장 성능이 좋다. 하지만 레지스터가 512개 라면 그림 13에서 나타나듯이 면적이 너무 커진다. 레지스터 512 개인 경우 128개와 비교했을 때 면적은 두 배 정도 차이가 난다. 하지만 예측한 실행시간은 512개의 레지스터에서 2%정도 성능향상이 있다. 실제로 비용을 고려한다면 128개의 레지스터를 사용했을 때 가장 적합한 선택이다.

5. 결 론

본 논문에서는 지오메트리 처리를 효율적으로 처리하면서 프로그램이 가능한 벡터 방식의 지오메트리 엔진

구조를 제안하였다. 또한 비슷한 H/W 비용을 갖는 Hitachi의 SH4와 시뮬레이션을 통해 VGE 구조의 효율성을 보였다.

VGE는 4개의 FADD, FMUL, 128개의 128비트 레지스터로 구성되며, 부동소수점 덧셈과 곱셈의 벡터연산, 벡터 단위의 스토어/스토어, 루프 언롤링을 이용하여 지오메트리 연산을 가속할 수 있다. 네 개의 부동 소수점을 저장할 수 있게 128비트 벡터 레지스터를 갖으며 덧셈과 곱셈에 있어서 벡터연산을 위해 버스의 폭도 128비트로 갖게 했다. 그리고 덧셈과 곱셈에 있어서 유연한 벡터연산을 위해 벡터 형 연산과 브로드 캐스트 형 연산을 지원했다. 또한 벡터 로드/스토어를 명령어를 지원하여 로드/스토어가 자주 발생하는 변환 연산에 성능향상을 꾀하였다.

VGE는 4개의 FADD, FMUL갖고서 변환 연산과 라이팅 연산을 모두 가속하였다. 특히 4개의 FADD와 FMUL을 분리하였기 때문에 이것을 합쳐서 Dot-product 연산기를 지원한 SH4 보다 많은 성능향상이 있었다. 또한 메모리 계층구조 관점에서도 실행시간을 예측하여 512개의 레지스터와 512 데이타 메모리 엔트리를 두었을 때 가장 좋은 성능을 보인다는 것을 보였다. 하지만 512개의 레지스터를 갖을 경우 128개의 레지스터를 갖을 때보다 면적이 2배가 커지지만 성능은 2% 밖에 좋아지지 않기 때문에 VGE는 128개의 레지스터를 갖도록 설계하였다.

비용을 고려할 때, VGE는 SH4와 비교해서 데이타 메모리와 레지스터를 합친 면적에 30%가 늘어났다 그러나 Viewperf 벤치마크 시뮬레이션 결과 SH4와 비교하여 평균 4.7배까지 성능향상을 보였다. 그러므로VGE는 저가형의 가격대비 성능이 뛰어난 지오메트리 구조이다.

참 고 문 헌

[1] Stuart Oberman, Greg Favor, and Fred Weber, "AMD 3DNow! Technology: Architecture and Implementations," IEEE Micro, pp. 37~48, 1999 March.

[2] nVIDIA Technology Brief, "Balancing Bottlenecks: The Partnership Between the CPU and GPU"

[3] Direct X Homepage, http://www.microsoft.com/directx/

[4] Neil Trevett, "PERMEDIA and GLINT Delta," In Proceeding Notebook for HOT Chips VIII, pp. 275~288, 1996 August.

[5] Neil Trevett, "GLINT Gamma: A 3D Geometry and Lighting Processor for the PC," In Proceeding Notebook for HOT Chips IX, pp. 235~246, 1997 August.

[6] Kurt Akeley, "Reality Engine Graphics," In Proceeding of SIGGRAPH '93, pp. 109~116.

[7] "Transform, lighting and rasterization system embodied on a single semiconductor platform," nVidia Patent, 1999 December.

[8] Fumio Arakawa and Osamu Nishii "SH4 RISC Multimedia Microprocessor," IEEE MICRO, pp. 26~34, 1998.

[9] M.Oka, and M.Suzuoki, "Designing and Programming the Emotion Engine," IEEE Micro, 1999 November.

[10] Alan E.Charlesworth and John L.Gustafson, "Introducing Replicated VLSI to Super computing: the FPS-164/MAX Scientific Computer," IEEE Computer, pp. 10~23, 1986.

[11] Neider, Mason and Tom Davis, "OpenGL Programming Guide," Addison & Wesley, pp. 197~215, 1997.

[12] Hennessy, John L and David A. Patterson, "Computer Architecture: A Quantitative Approach," Morgan Kaufmann, pp. 221~238, 1990.

[13] SPECviewperf benchmark, http://www.spec.org/gpc/opc.static/opcview.htm

김 원 석

2000년 홍익대학교 컴퓨터 공학과 졸업(학사). 2002년 연세대학교 대학원 컴퓨터과학과 졸업(공학석사). 2002년~현재 대우 일렉트로닉스 리빙연구소 리빙회로팀 연구원. 관심분야는 고성능 컴퓨터구조, 3차원 그래픽 가속기, HCI

정 철 호

1996년 숭실대학교 소프트웨어공학과 졸업(학사). 1998년 연세대학교 대학원 컴퓨터과학과 졸업(공학석사). 1998년~현재 연세대학교 대학원 컴퓨터과학과 박사과정 재학중. 관심분야는 고성능 컴퓨터구조, 3차원 그래픽 가속기, Computer Arithmetic, ASIC 설계

한 탁 돈

1978년 연세대학교 공과대학 전자공학과 졸업(학사). 1983년 Wayne State University 컴퓨터공학(공학석사). 1987년 University of Massachusetts 컴퓨터공학(공학박사). 1987년~1989년 Cleveland 주립대학 조교수. 1989년~현재 연세대학교 공과대학 컴퓨터과학과 교수. 관심분야는 Wearable computer, HCI, ASIC 설계, 고성능 컴퓨터구조