멀티코어 GP-GPU를 이용한 지오메트리 처리

Geometry Processing using Multi-Core GP-GPU

이 광 엽*, 김 치 용*★
Kwang-Yeob Lee*,Chi-Yong Kim*★

Abstract

A 3D graphics pipeline is largely divided into geometry stage and rendering stage. In this paper, we propose a method that accelerates a geometry processing in multi-core GP-GPU, using dual-phase structure. It can be improved by parallel data processing using SIMD of GP-GPU, dual-phase structure and memory prefetch. The proposed architecture improves approximately 19% of performance when it use all the features.

요 약

3D 그래픽 처리 과정은 크게 지오메트리 단계와 렌더링 단계로 구분된다. 본 논문에서는 듀얼페이즈 멀티코어 GP-GPU에서 지오메트리 처리를 가속화시키기 위한 방법을 제안한다. GP-GPU의 SIMD, 듀얼페이즈 구조를 이용한 병렬적 데이터 처리와 메모리 프리패치를 이용하여, 지오메트리 처리를 가속화 시킬 수 있었으며, 모든 기능을 사용할 시 19%의 성능 향상을 나타내었다.

Key words : Dual-Phase, SIMD, Prefetch, Geometry, GP-GPU

Ⅰ. 서론

최근 모바일 환경에서도 화려한 GUI(Graphic User Interface)나 다양한 그래픽 효과를 이용한 멀티미디어 컨텐츠들이 요구됨에 따라 그래픽 처리를 위한 CPU의 부담이 커지고 있다. 따라서, 이러한 CPU의 부담을 덜고 전력소모를 최소화하기 위하여 모바일 기기에도 GPU(Graphic Processing Unit)의 탑재가 필요조건이 되었다.

과거의 GPU는 고정된 파이프라인을 통하여 한정된 그래픽 처리만이 가능하였다. 하지만 Shader가 등장함에 따라 GPU의 프로그래밍 렌더링 파이프라인을 통해 다양한 효과와 고화질의 그래픽 처리가 가능하게 되었다. 또한 GPU는 그래픽 처리를 위해 연산 부분에 최적화되어 CPU보다 빠르게 발전하고 있으며, 이러한 GPU의 연산 능력을 활용하여 그래픽 처리뿐만 아니라 CPU에 많은 연산이 필요한 경우, 이를 나누어 처리할 수 있는 GP-GPU(General-Purpose computing on Graphics Processing Units) 구조로 발전하고 있다.[1]

본 논문에서는 모바일 환경에 적합하도록 설계된 GP-GPU를 이용하여, 3D 그래픽 처리 과정 중 지오메트리(geometry) 파트를 구현하였다. 본 논문에서 사용된 GP-GPU는 멀티스레드(Multi-Thread)와 듀얼페이즈(Dual-Phase)의 가변길이 명령어 구조를 채택하고 있으며, 이렇게 설계된 멀티스레드 싱글코어를 최대 16까지 멀티코어로써 동작 가능하도록 설계되었다.[2][3] 프로세서는 멀티코어, 멀티스레드 구조이기 때문에 그래픽 처리 데이터의 병렬성을 활용하여 높은 처리 효율을 나타낼 수 있다. 또한, GP-GPU 구조이기 때문에 별도의 CPU나 H/W 추가 없이, 프로그래밍을 통하여 하나의 IP로 모든 동작의 지원이 가능하며, 알고리즘의 개선으로 성능 향상이 가능하다. 구현된 지오메트리는 상기의 프로세서가 가진 메모리

*정회원,서경대학교 컴퓨터공학과 교수
*★ 정회원,서경대학교 컴퓨터과학과 교수, 교신저자
※ 본 논문은 서울시 산학연 협력사업(10560)의 지원으로 작성되었으며 설계에는 IDEC의 지원장비로 이루어졌습니다.
接受日:2010年 5月 29日, 修正完了日: 2010年 6月 25日

프리패치와 SIMD, 듀얼페이즈 구조를 활용하여 효율적인 처리가 가능하였다.

II. 본론

1. 멀티코어 GP-GPU 구조

본 논문에서는 4-코어 GP-GPU를 이용하여 검증을 진행 하였다. 프로세서의 기본적인 구조는 그림 1과 같으며, 싱글코어를 기준으로 살펴보면 크게 코어와 레지스터 모듈로 구분할 수 있다.

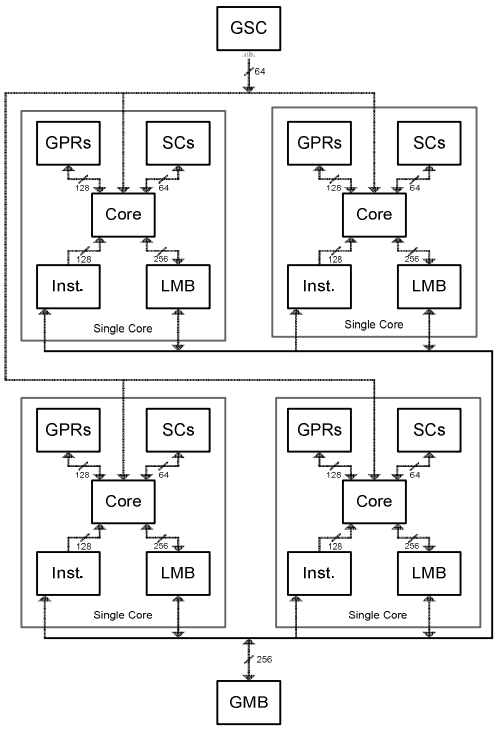


그림 1. 4-코어 GP-GPU 구조
Fig 1. Structure of a 4-core GP-GPU

코어는 12개의 스레드들이 round-robin 방식을 통해 순차적으로 실행되며, 듀얼페이즈 구조를 적용하여 2개의 Phase를 통해 각 Phase로 입력된 명령어들을 동시에 처리 할 수 있도록 설계되었다. 이것은 하나의 ALU를 사용하지만 동시에 두 개의 서로 다른 연산기를 사용함으로써 처리 속도를 높이는 기법이다. 그렇기 때문에, 양 페이즈에서 동일한 연산기를 사용할 수 없는 제약이 발생하지만, 그 외의 경우에서는 처리 성능을 더욱 높일 수 있다.

레지스터 모듈은 코어의 연산을 위해 필요한 레지스터들로써, 코어에 입력되기 위한 명령어들을 저장하는 Inst.(Instruction Bank), 코어에서 각각 독립적으로 수행되는 Thread들이 데이터를 read/write 하기 위한 GPRs(General Purpose Registers), 모든 Thread가 공유하는 Shared Memory 역할을 하는 LMB(Local Memory Block)과 SCs(Scratch Counter)로 구성 된다.

이러한 싱글코어들 간의 데이터를 공유하기 위해 GMB(Global Memory Block)과 GSC(Global Scratch Counter)를 사용하여 멀티코어를 구성한다.

2. 지오메트리 알고리즘

지오메트리 처리는 그림 2와 같은 순서로 진행된다. 우선 첫 번째 정점을 입력 받아 Vertex Shading을 수행한 후, 이 과정을 3번 반복하여 폴리곤을 생성한다. 이후 컬링을 수행한 다음, 제거되지 않은 폴리곤에 대해 저장을 한다. 이러한 과정을 모든 폴리곤에 대해 수행을 하고 종료를 하게 된다.

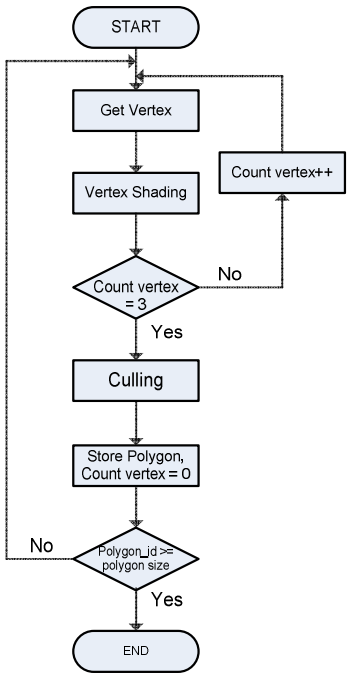


그림 2. 지오메트리 처리 순서도
Fig 2. Flow chart of a Geometry Process

가. 메모리 유휴시간 제거

4-코어 구조에서, 1개의 명령어는 실제적으로 4(Cores) * 12(Threads)개 만큼 수행된다. 메모리 명령어의 경우, 명령어 자체의 수행 시간이 긴 것뿐만

아니라, 하나의 버스를 통하여 데이터를 전송해야 하므로, 순차적 처리로 인해 소요 시간은 더욱 증가하게 된다. 메모리 명령어는 그림 3과 같이 각 코어의 Queue에 저장되고, 총 48개의 명령어가 순차적으로 수행된다. 메모리 명령어가 처리되기 위한 시간을 측정 해 본 결과 스레드당 200 clock 정도의 시간이 소요되는 것을 확인 할 수 있었다. 이러한 메모리 유휴시간은 성능 감소로 이어지게 되고, 메모리 명령어가 많을수록 그 영향이 심각해진다.

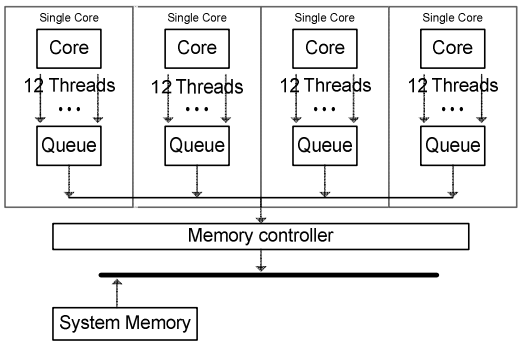


그림 3. 메모리 명령어 처리 구조
Fig 3. Structure of a Memory Instruction Process

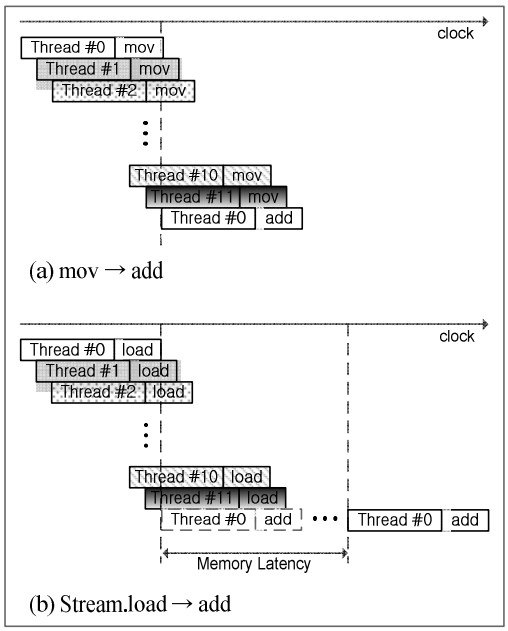


그림 4. 메모리 명령어에 의한 유휴시간
Fig 4. Stall by memory instructions

그림 4는 명령어에 따른 수행시간의 차이를 나타낸다. 메모리 명령어를 제외한 일반적인 명령어의 동작은 그림 4(a)와 같다. Thread #0를 살펴보면 mov 명령어가 완료된 시점에서 add 명령어가 수행되는 것을 알 수 있다. 그러나 그림4(b)와 같이 load와 add 명령어가 수행되는 경우에는 지연시간이 발생한다. 이것은 add 명령어가 수행되기 전에 처리될 데이터의 전송이 완료 되어야 하기 때문이다. 결국, 메모리 명령어가 완료 될 때까지 다음 명령어가 수행 될 수 없기 때문에, 프로세서 코어는 스레드의 동작을 멈추고, 메모리 전송이 끝난 이후에 add명령어를 수행한다.

이러한 현상을 제거하거나 우회하는 방법을 메모리 유휴시간 제거 (Memory Latency Hiding) 기능이라고 한다. 본 논문에서는 메모리 전송에 의한 유휴시간을 제거하기 위해 메모리 프리패치 기능을 활용하였다.

메모리 명령어는 Stream.load와 Stream.store로 나뉜다. 이 명령어에 그림 5와 같이 prefetch와 bypass 옵션을 사용한다. prefetch와 bypass 옵션 모두 곧바로 다음 명령어가 실행되도록 한다. 하지만 prefetch의 경우 Stream.Wait 명령어를 통해 메모리 전송의 완료 여부를 확인하여 기다릴 수 있는 기점을 지정할 수 있다.

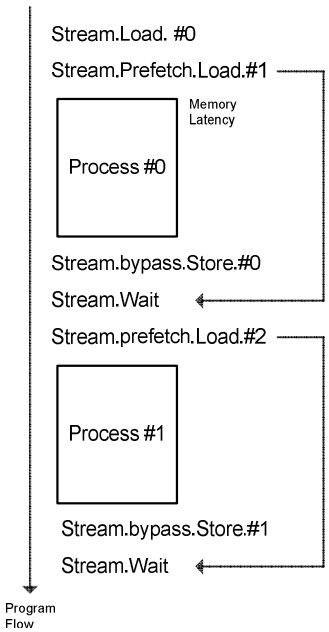


그림 5. 메모리 프리패치에 의한 메모리 유휴시간 활용
Fig 5. Compensation for a memory stall by a memory prefetch

그림 5의 동작을 살펴보면, 우선 Stream.Load.#0를 통해 Process #0에서 처리할 데이터들을 메모리로부터 읽어온다. 이때는 처리할 수 있는 데이터가 아무것도 없기 때문에 첫 번째 load 명령어에 의한 유휴 시간 제거는 불가능하다. #0의 메모리 전송이 끝난 이후, Stream.Prefetch.Load.#1 명령어를 통해 Process #1에서 처리할 데이터들을 읽어온다. prefetch 옵션을 사용하였으므로, 프로세서는 메모리 전송 완료를 기다리지 않고, 바로 다음 명령어를 수행한다. 즉, Process #1에서 사용될 데이터의 전송과 Process #0에 해당하는 명령어들의 처리가 동시에 이루어진다.

Process #0의 동작이 완료된 이후, Stream.bypass.Store.#0 명령어로 Process #0의 연산 결과를 메모리에 저장한다. bypass 옵션을 사용하였으므로 prefetch와 마찬가지로 바로 다음 명령어를 수행한다. 다음 명령어인 Stream.Wait는 prefetch의 완료를 기다리는 명령어이다. Stream.Wait가 실행되는 시점까지 Stream.Prefetch.Load.#1의 전송이 완료되지 않았다면, 이를 기다린 후 다음 명령어를 수행하고, 전송이 완료되었다면 바로 다음 명령어를 수행한다.

마찬가지로 다음 동작들은 Stream.Prefetch.Load.#2 명령어로 Process #2에서 사용될 데이터들을 전송하고, Prefetch와 Wait를 통해 전송이 완료된 Process #1에 대한 명령어들을 수행하게 된다. 따라서, 이 방법을 통하여 다음에 사용할 데이터를 먼저 요청하고 메모리 유휴 시간 동안 현재 데이터를 처리하는 방법으로 메모리 유휴 시간을 제거할 수 있다.

Prefetch를 사용한다고 하더라도 현재 데이터의 처리 시간이 짧다면 높은 효율을 나타내지는 않는다. 앞서 말했듯이, 메모리 전송이 200 clock 정도 소요된다고 하고, 데이터 처리가 10 clock 정도라고 가정하면 총 수행 시간은 다음과 같다.

Prefetch : 200 clock
(동시 수행으로 10 clock 포함)
Non-Prefetch : 200 clock + 10 clock = 210 clock
(메모리 전송 이후, 데이터 처리)

이때는, Prefetch의 적용 효과가 5%로 미미하게 발생한다. 하지만 데이터 처리 시간이 긴 경우에 높은 효율을 나타낸다. 예를 들어 데이터 처리 시간이 메모리 전송과 같은 200 clock 이 걸린다고 하면 이 경우에는 50%의 성능 향상을 나타낼 수 있다.

지오메트리 처리에서는 인덱스 배열과 정점 데이터에 대해 적용할 수 있다. 인덱스 배열에 대해서는, 메모리로부터 읽어오는 동작을 prefetch로 수행하고 그 동안 시스템 초기화 동작들을 수행하도록 한다. 정점 데이터에 대한 prefetch는, 우선 다음에 처리할 정점 데이터를 메모리에서 각각의 GPRs로 읽어오는 동작을 prefetch로 수행하고, 그 동안 현재 처리할 정점 데이터에 대한 연산을 수행한다.

나. 데이터 병렬성

지오메트리 처리는 대부분 좌표의 변환으로, 행렬 곱셈을 수행하게 된다. 그림 6과 같이 행렬 곱셈은 16번의 곱셈과 12번의 덧셈으로 이루어지므로, 일반적인 프로세서에서는 총 28번의 명령어가 수행되어야 한다.

$$\begin{bmatrix} m0 & m1 & m2 & m3 \\ m4 & m5 & m6 & m7 \\ m8 & m9 & m10 & m11 \\ m12 & m13 & m14 & m15 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

$$T.x = (m0 \times x) + (m1 \times y) + (m2 \times z) + (m3 \times w)$$
$$T.y = (m4 \times x) + (m5 \times y) + (m6 \times z) + (m7 \times w)$$
$$T.z = (m8 \times x) + (m9 \times y) + (m10 \times z) + (m11 \times w)$$
$$T.w = (m12 \times x) + (m13 \times y) + (m14 \times z) + (m15 \times w)$$

그림 6. 좌표변환을 위한 행렬 곱셈
Fig 6. Matrix multiplication for a coordination transform

본 논문의 GP-GPU는 기본적으로 3D 그래픽을 효과적으로 구현하기 위해 4way SIMD 구조를 취하였다. 따라서, x,y,z,w의 4D Vector에 대하여 하나의 명령어로 동시 연산이 가능하다. 또한 듀얼페이즈 구조를 통하여 각 페이즈로 입력된 서로 다른 명령어를 동시에 처리 할 수 있기 때문에, 행렬 연산 중 중복되지 않는 서로 다른 연산에 대해 동시 실행이 가능하여 연산 속도를 높일 수 있다.

(1) SIMD 구조

하나의 명령어로 4D Vector를 한 번에 처리할 수 있기 때문에 연산 횟수를 줄일 수 있다. 그림 6의 아래 식을 보면 m0, m4, m8, m12는 각각 x와 곱해지고 다른 식들과 데이터의 의존성이 없는 것을 알 수 있다. 따라서, 한번에 4개의 원소를 x와 곱하여 계산이 가능하다. 마찬가지로 y, z, w에 대해서도 해당하는 각 원소들을 한 번에 곱하여 계산할 수 있다.

SIMD구조를 이용한 행렬 연산은 그림 7과 같다. 우선 각 x, y, z, w와 곱해지는 행렬의 원소들을 정렬하여 4D Vector의 곱셈을 수행하고(①×x, ②×y, ③×z, ④×w), 그 결과에 대하여 4D Vector 덧셈을 수

행한다(⑤+⑥+⑦+⑧). 최종결과인 ⑨~⑫를 보면 그림 6의 연산 결과와 같은 것을 확인할 수 있다. 그 결과, 4개의 곱셈 명령어와 3개의 덧셈 명령어, 즉 총 7개의 명령어만으로 행렬 연산이 가능하기 때문에 수행 시간을 단축할 수 있다.

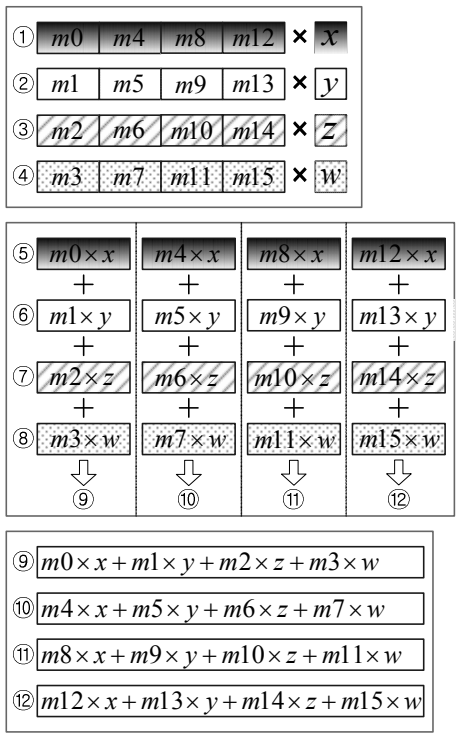


그림 7. SIMD구조를 이용한 행렬 곱셈
Fig 7. Matrix multiplication using a SIMD structure

(2) 듀얼페이즈 구조

앞서 설명한 것처럼 본 논문의 프로세서는 서로 다른 연산에 한하여 동시에 처리가 가능하므로, 행렬 연산의 곱셈과 덧셈 연산을 동시에 수행함으로써 연산에 필요한 명령어를 더욱 단축시킬 수 있다.

이 연산 과정을 아래에 나타내었다. r[]은 128bit 레지스터를 의미하며, []안의 번호는 해당 레지스터의 인덱스이다. r[]의 접미사 .x는 128bit의 레지스터 중 x에 해당하는 32bit 데이터를 사용하겠다는 의미이다.

MAT_0 = m0, m4, m8, m12
MAT_1 = m1, m5, m9, m13
MAT_2 = m2, m6, m10, m14
MAT_3 = m3, m7, m11, m15

```
r[0] = r[V_IN].x * r[MAT_0];
r[1] = r[V_IN].y * r[MAT_1];
r[2] = r[V_IN].z * r[MAT_2] / r[0] += r[1];
r[3] = r[V_IN].w * r[MAT_3] / r[0] += r[2];
r[V_OUT] = r[0] + r[3];
```

곱셈과 덧셈이 동시에 가능하지만, 행렬 연산에서 덧셈은 데이터 종속성이 있으므로, 곱셈 연산 이후에 수행되어야 한다. 그러므로, (x * MAT_0)와 (y * MAT_1)가 먼저 수행된 이후, (z * MAT_2)가 수행될 때, (x * MAT_0)와 (y * MAT_1)의 결과를 더한다. 이 방식으로 2개의 명령어를 추가로 줄일 수가 있다.

III. 성능

구현된 각각의 기능들을 검증하기 위해, 그림 8의 모델 데이터들을 이용하여 지오메트리 처리를 하였다. 검증 시스템은 Test Drive를 이용하였으며, 100MHz로 설정하여 시뮬레이션 하였다.[4]

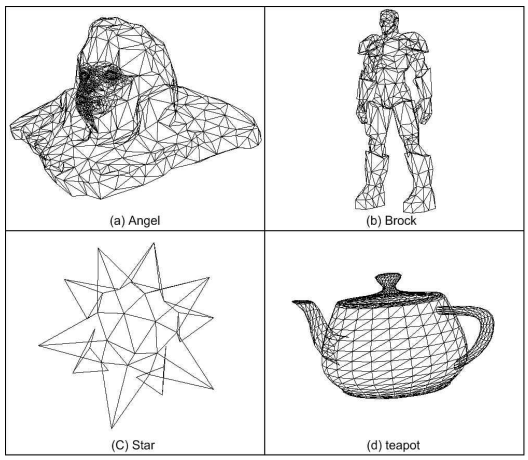
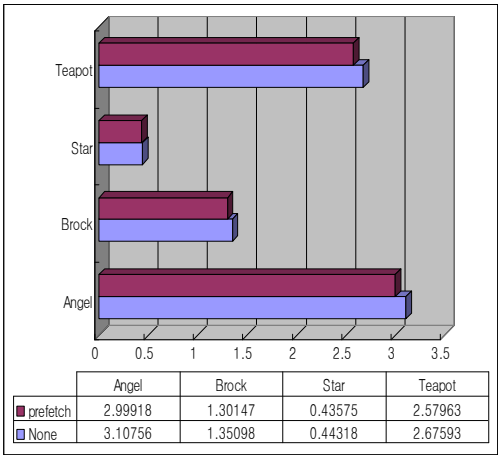


그림 8. 검증 모델
Fig 8. Verification Models

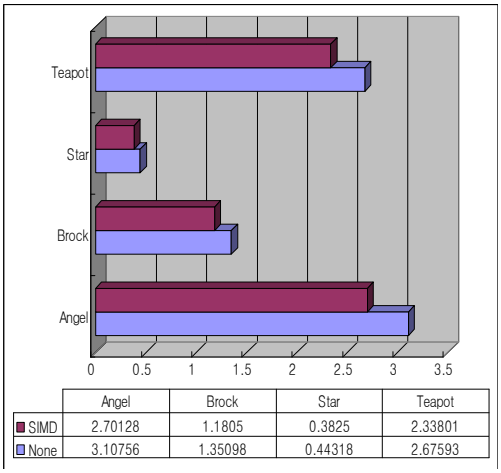
각 기능별로 4가지 모델 데이터의 시뮬레이션 수행 결과를 그래프로 나타내었으며, x축은 수행 완료까지의 경과시간(milli second)이다.



	Angel	Brock	Star	Teapot
prefetch	2.99918	1.30147	0.43575	2.57963
None	3.10756	1.35098	0.44318	2.67593

그림 9. 메모리 프리패치 적용 결과
Fig 9. Applied results to memory prefetch

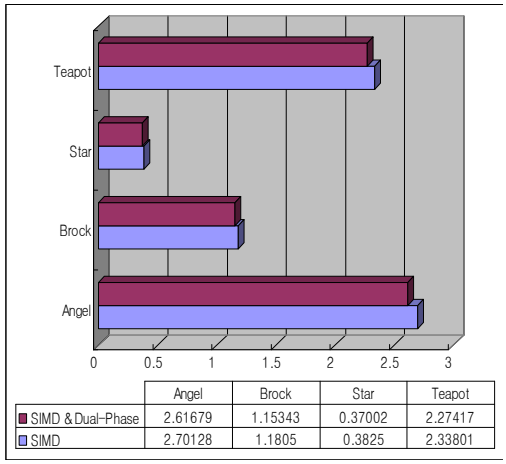
그림 9는 메모리 프리패치 기능을 사용한 경우의 성능을 나타내는 그래프이다. None은 아무런 기능도 적용하지 않은 경우이다. 프리패치를 적용한 경우, 평균 3.1%의 속도 향상을 나타내었다.



	Angel	Brock	Star	Teapot
SIMD	2.70128	1.1805	0.3825	2.33801
None	3.10756	1.35098	0.44318	2.67593

그림 10. 4D Vector 연산 결과
Fig 10. Results of a 4D Vector arithematic

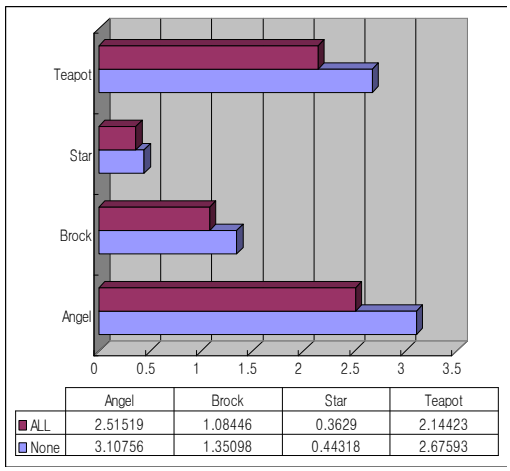
그림 10은 SIMD 구조를 이용한 4D Vector 연산을 사용한 경우의 성능을 나타내는 그래프이다. 기존의 처리 방법에 비해 약 13%의 성능 향상을 나타내었다.



	Angel	Brock	Star	Teapot
SIMD & Dual-Phase	2.61679	1.15343	0.37002	2.27417
SIMD	2.70128	1.1805	0.3825	2.33801

그림 11. 듀얼페이즈 적용 결과
Fig 11. Applied results to dual phase

그림 11은 SIMD 구조에서 Dual-Phase를 적용한 경우의 성능을 나타내는 그래프이다. Dual-Phase 연산은 Vertex Shading에서의 행렬 곱셈 뿐 만 아니라 이외의 적용 가능한 몇몇 부분에도 사용하여 지오메트리 처리를 하였다. 그 결과, 약 2.9%의 성능 향상을 나타내었다.



	Angel	Brock	Star	Teapot
ALL	2.51519	1.08446	0.3629	2.14423
None	3.10756	1.35098	0.44318	2.67593

그림 12. 모든 기능을 적용한 결과
Fig 12. Applied results to every way

그림 12는 메모리 프리패치와 SIMD 구조를 이용한 4D Vector 연산, 그리고 Dual-Phase를 모두 적용한 경우와 기본적인 지오메트리 처리 방법과의 성능을 비교한 그래프이다. 모든 기능은 사용한 경우 기

존 대비 약 19%의 성능 향상을 나타내었다.

Ⅳ. 결론

본 논문에서는 멀티코어 GP-GPU에 대해 살펴보고 이를 이용한 지오메트리 처리 방법을 제안하였다. 처리 속도를 높일 수 있는 방안 중 하나로써 메모리 프리패치 기능을 이용하여 메모리 유휴시간을 줄일 수 있었다. 또한 GP-GPU의 SIMD와 듀얼페이즈 구조를 이용한 병렬적 데이터 처리를 이용하여 성능을 좀 더 상향시킬 수 있었다. HDL 시뮬레이터를 이용하여 지오메트리 처리를 수행한 결과, 모든 기능을 사용할 시 약 19%정도의 성능 향상을 보였다.

참고문헌

[1] Mauricio Breternitz, Jr., "Compilation, Architectural Support, and Evaluation of SIMD Graphics Pipeline Programs on a General-Purpose CPU", Proceedings of the 12th international conference on parallel architectures and compilation techniques.

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

[3] H.K. Jeong, "A Multi-thread Processor Architecture With Dual Phase Variable-Length Instructions" ITC-CSCC 2008

[4] H.K. Jeong, "Test-Drive System for a Design & Verification of a GP-GPU Processor" 2010 SoC Conference

저 자 소 개

이 광 엽(정회원)

1985년 8월 서강대학교 전자공학과 학사.
1987년 8월 연세대학교 전자공학과 석사.
1994년 2월 연세대학교 전자공학과 박사.
1989~1995년 현대전자 선임연구원.
1995년~현재 서경대학교 컴퓨터공학과 부교수.

<주관심분야 : 마이크로 프로세서, Embedded System, 3D Graphics System>

김 치 용 (정회원)

1981년 2월 경북대학교 통계학과 학사.
1986년 2월 서울대학교 계산통계학과 이학석사.
1993년 2월 서울대학교 계산통계학과 이학박사.
1995년~현재 서경대학교 컴퓨터과학과 부교수.

<주관심분야 : Stochastic process, Mathematical analysis, Computer arithematic>