Journal of Korea Multimedia Society Vol. 14, No. 5, May 2011(pp. 669-680)

GPU를 이용한 R-tree에서의 범위 질의의 병렬 처리

유보선[†], 김현덕[††], 최원익[†††], 권동섭[††††]

요 약

R-tree는 데이터베이스 시스템에서 가장 많이 사용되는 색인 구조로 다차원의 데이터를 관리하는데 매우 효율적이다. 하지만 데이터베이스 시스템이 처리해야 하는 데이터의 용량이 증가함에 따라, 기존의 R-tree에서의 범위 질의의 처리는 디스크의 접근 지연 등의 이유로 인하여 수행 시간이 증가하게 되었다. 이러한 문제들을 해결하기 위하여 버퍼를 사용하거나 혹은 다수의 디스크와 프로세서를 사용하여 병렬로 질의를 수행하고자 하는 많은 연구들이 진행되었다. 이러한 연구들의 일환으로 최근 Graphics Processing Unit (GPU)을 이용한 병렬화 기법들에 대한 연구들이 진행되고 있다. 이러한 GPU의 적용을 통한 병렬화는 계산 속도의 증가와 디스크 접근 횟수의 감소를 통하여 수행 속도의 개선을 가능하게 하지만 GPU와 CPU사이의 메모리 교환 및 GPU 메모리의 접근 지연 등에 의한 오버헤드를 발생시킨다. 본 논문에서는 이러한 오버헤드를 해결하고 효과적으로 GPU를 적용하기 위하여, GPU를 버퍼로 사용하여 범위 질의를 병렬화하는 기법을 제안하였다. 버퍼 알고리즘을 통하여 메모리 교환 횟수를 줄이고, 동시 접근 가능한 메모리의 용량을 증가시켜 메모리의 접근 지연을 최소화 할 수 있었다. 제안 기법과 기존의 인덱스의 비교 실험에서 최대의 경우 5배 정도의 성능이 개선되는 것을 확인 할 수 있었다.

Parallel Range Query processing on R-tree with Graphics Processing Units

Boseon Yu[†], Hyunduk Kim[††], Wonik Choi[†††], Dongseop Kwon[††††]

ABSTRACT

R-trees are widely used in various areas such as geographical information systems, CAD systems and spatial databases in order to efficiently index multi-dimensional data. As data sets used in these areas grow in size and complexity, however, range query operations on R-tree are needed to be further faster to meet the area-specific constraints. To address this problem, there have been various research efforts to develop strategies for acceleration query processing on R-tree by using the buffer mechanism or parallelizing the query processing on R-tree through multiple disks and processors. As a part of the strategies, approaches which parallelize query processing on R-tree through Graphics Processor Units (GPUs) have been explored. The use of GPUs may guarantee improved performances resulting from faster calculations and reduced disk accesses but may cause additional overhead costs caused by high memory access latencies and low data exchange rate between GPUs and the CPU. In this paper, to address the overhead problems and to adapt GPUs efficiently, we propose a novel approach which uses a GPU as a buffer to parallelize query processing on R-tree. The use of buffer algorithm can give improved performance by reducing the number of disk access and maximizing coalesced memory access resulting in minimizing GPU memory access latencies. Through the extensive performance studies, we observed that the proposed approach achieved up to 5 times higher query performance than the original CPU-based R-trees.

Key words: Database(데이터베이스), Index Structure(색인구조), Paralle Processing(병렬처리)

※ 교신저자(Corresponding Author) : 최원익, 주소 : 인하대학교 하이테크관 511호, 전화 : 032)860-8375, FAX : 032)873-8970, E-mail : wichoi@inha.ac.kr
접수일 : 2010년 10월 22일, 수정일 : 2011년 1월 6일
완료일 : 2011년 5월 5일
[†] 준회원, 인하대학교 일반대학원 정보공학과
(E-mail : man82bs@gmail.com)
[††] 준회원, 인하대학교 일반대학원 정보공학과
(E-mail : force80@naver.com)
[†††] 정회원, 인하대학교 정보통신공학부 조교수
[††††] 정회원, 명지대학교 컴퓨터공학과 부교수
(E-mail : dongseop@gmail.com)

1. 서 론

Graphics Processing Units(GPU)은 CPU의 그래픽 처리 작업을 보조하기 위한 장치로 고도 병렬처리(massively parallel processing)에 적합하며, 단일 명령어로 다중의 데이터를 처리할 수 있는 구조(SIMD, Single Instruction Multiple Data)를 가지고 있어서 병렬 처리에 매우 유리한 구조를 가지고 있는 하드웨어 플랫폼이다[1]. Geforce GTX 480 그래픽카드의 경우, 15개의 멀티프로세서를 지원하며, 각 멀티프로세서가 32개의 프로세서를 가지고 있다. 이 그래픽 카드는 초당 1300 GFLOPS의 연산이 가능하며, 180GB/s의 대역폭을 가지고 있다[2]. 이러한 GPU의 뛰어난 계산 능력과 병렬 처리 능력 그리고 빠른 데이터 전송 등의 이점으로 인해서 범용 목적(General Purpose computing on Graphics Processing Units)으로 이미 다양한 분야에서 활용되고 있다. 본 논문에서는 이러한 GPU를 이용하여 R-tree에서의 범위 질의의 수행 속도를 개선하기 위한 방법을 제안한다. R-tree[3]는 공간자료 데이터베이스 시스템에서 가장 널리 사용되는 자료구조로서, B-tree에서 확장되었다. R-tree는 다차원의 자료를 관리하는데 매우 적합하며, 공간 데이터에 대한 삽입, 삭제, 검색 등의 작업을 효율적으로 수행할 수 있는 구조를 가지고 있는 색인 구조이다. R-tree에서의 검색은 root로부터 말단 노드까지 재귀적으로 순회하며 수행된다. 이러한 검색은 상당히 효율적이긴 하지만 GIS, CAD와 같이 현대의 대용량화되는 데이터베이스에서는 상당한 수행시간을 요구한다. 특히, 범위 질의와 같이 특히 더 많은 노드들을 차례로 순회해야 하는 경우에는 더 긴 수행시간을 필요로 한다. 이러한 R-tree의 성능을 개선하기 위하여 R-tree의 병렬화에 대한 기법들이 연구되었다. R-tree를 병렬화하는 기존의 기법들은 단일 워크스테이션에서 다수의 디스크와 프로세서들을 사용하여 동시 디스크 입출력을 통하여 처리하거나[4-6] 혹은 워크스테이션들을 네트워크로 연결하여 사용하는 방식[7-10], GPU에 의한 방식 등이 있다. 그러나 이러한 기법들은 다수의 디스크 혹은 워크스테이션들을 사용함으로써 많은 비용을 필요로 하고, 다수의 디스크에서 동시 입출력을 사용하는 것에서 알 수 있듯이, 전체 데이터를 여러 디스크에 분배하는 등의 오버헤드가 존재한다. 또한 어떻게 데이터가 분포하는가에 따라서 범위 질의의 성능이 달라진다.

본 논문에서는 GPU를 트리 형태의 색인 구조에 적용하여 범위 질의를 처리하는 기법을 제안한다. GPU는 Grid File[11]과 같이 수평적 구조를 가지고 있는 색인 구조에는 적용하기가 상대적으로 용이하나 트리와 같이 수직적인 구조를 가지고 있는 색인구조에는 적용하기가 쉽지 않다. 왜냐하면, 색인 기법들은 많은 디스크 접근과 상대적으로 적은 계산량을 필요로 하는 기능을 담당하고 있으며, 적용을 하더라도 각 노드가 실제로 비교될 key들을 가지고 있으며, 그 노드들이 연속한 메모리 공간에 존재하지 않기 때문이다. 이것은 결국 CPU와 GPU사이의 저용량의 잦은 메모리 이동을 야기한다. 이러한 문제점을 해결하기 위하여, 본 논문에서 제안하는 알고리즘은 GPU의 global 메모리를 버퍼로 사용하며, GPU의 프로세서들을 이용하여 병렬로 검색함으로써, R-tree에서의 검색 성능을 향상시키도록 한다. 본 논문에서 제안하는 기법은 기존의 R-tree에 대한 수정을 최소화 하여 버퍼링되지 않은 부분에 대한 검색의 효율성을 유지하였다. 대략적인 알고리즘은 다음과 같다. 수행된 질의를 저장하고, 질의가 방문한 말단 노드들의 엔트리들을 GPU 메모리에 저장하여, 다음 질의 수행 시 검색이 R-tree와 GPU에 의해서 동시에 수행되도록 한다. 따라서 수행되는 질의가 요구하는 말단 노드들이 버퍼에 얼마나 존재하는가에 따라서 성능의 향상 정도가 결정된다. 본 논문의 구성은 다음과 같다. 1장의 서론에 이어서 2장에서는 GPU의 적용에 의해 발생하는 오버헤드에 대하여 살펴본다. 3장에서는 기존의 병렬 색인 기법에 대하여 제시하였으며, 4장에서는 GPU를 이용한 효율적인 범위 질의 알고리즘을 제안하며, 5장에서는 제안하는 알고리즘과 R-tree의 검색 성능을 비교하기 위한 실험 결과를 제시하고 마지막 장에서 결론을 맺는다.

2. GPU

2.1 GPU overhead

GPU를 R-Tree와 같이 계층 적인 인덱스 구조에 적용할 경우, 계산 속도의 증가를 가져올 수 있으나, 우선 GPU의 적용에 의한 여러 오버헤드들을 고려해야 한다. GPU에 의한 주된 오버헤드는 CPU와 GPU

사이의 메모리 교환과 GPU가 계산을 수행할 때의 메모리 접근 등에서 발생한다. 특히, 메모리 교환에 의해서 발생하는 오버헤드가 상대적으로 크다.

그림 1은 CPU와 GPU 사이의 메모리 교환 시 소모되는 시간을 메모리의 양을 증가시키면서 실험한 것이다. 그림 1에서 알 수 있듯이, 교환하는 용량에 비례하여 시간이 증가하며, 같은 크기의 메모리 전송 시, CPU 에서 GPU로의 메모리 이동보다 GPU에서 CPU로의 메모리 이동이 대략 2, 3배 이상의 시간을 소모한다는 것을 알 수 있다. 메모리 교환은 GPU에 계산을 요청할 때와 GPU가 계산을 완료하여 그 결과를 가져올 때 각각 발생한다. 따라서 그림 1은 질의 수행 후 결과를 다시 CPU로 가져올 때 발생하는 부하가 비교 요청 시 발생하는 부하보다 크다는 것을 보여준다. 따라서 메모리 교환에 의한 오버헤드를 최소화하기 위해서는 결과 셋의 용량을 최소화 하여야 한다. 그림 2는 CPU 와 GPU가 MBR을 질의의 범위와 비교하는 시간을 각각 측정한 것이다. 메모리 접근에 의한 오버헤드를 최소화하기 위하여 global memory에 대한 접근 횟수를 최소화하고 동시 접근 횟수를 최대화함으로써 오버헤드를 최소화할 수 있다. GPU의 수행시간은 칩 외부에 존재하여 응답 속도가 느린 global 메모리의 접근시간을 포함하고 있기 때문에 적은 수의 MBR을 비교하는 경우, CPU의 경우 보다 더 오랜 수행 시간을 필요로 한다. 그림 2에서 볼 수 있듯이, 임계치 T개 이하의 MBR을 비교하는데 필요로 하는 시간은 오히려 CPU보다 크다

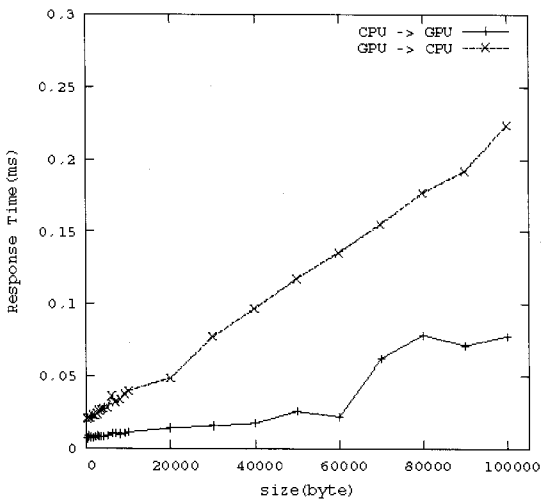

그림 1. 메모리 교환에 의한 오버헤드

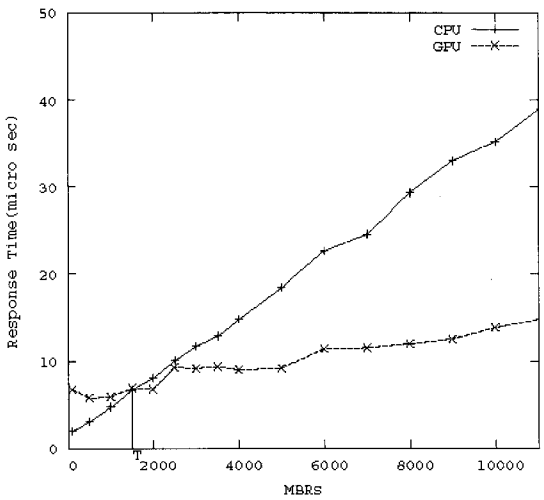

그림 2. GPU와 CPU에 의한 MBR 비교 시간의 비교 (Global memory access)

는 것을 알 수 있다. 비교하는 MBR의 수가 T개를 초과하면서 성능이 역전되기 시작하며, MBR의 개수가 증가할수록 뚜렷한 계산 속도의 차이가 있음을 알 수 있다. 따라서 GPU를 이용하기 위한 오버헤드를 최소화하기 위해서는, 질의의 선택도를 T 이상으로 선택하고, GPU와 CPU사이의 메모리 교환 횟수 및 용량을 최소화해야 한다. T는 실험환경에 따라서 다를 수 있는데 본 논문에서는 대략 1500개의 MBR을 비교 했을 때, 비슷한 수행 속도가 측정되었다.

2.2 R-tree의 GPU 적용

데이터베이스 시스템은 방대한 양의 자료를 처리하기 위하여 디스크 기반으로 동작한다. 즉 GPU를 데이터베이스 시스템에 적용하기 위해서는 처리되어야 할 데이터를 디스크로부터 읽어 와서 GPU로 전달해 주어야 한다. 데이터베이스에서의 디스크 접근은 사용되는 색인 구조에 따라서 접근 횟수 혹은 방식 등이 달라진다. R-tree와 같은 색인 구조는 하나의 노드가 하나의 디스크 페이지와 연결되어 동작하며, 한 페이지에 저장될 수 있는 데이터의 양은 노드의 최대 entry 수 m값과 활용률에 의해서 제한된다. 또한 R-tree의 계층적인 구조에 의해서 노드 사이의 이동은 페이지 offset에 의하여 직접 이루어지기 때문에 관계되는 노드를 순차적으로 방문해야만 원하는 말단 노드까지의 이동이 가능하다. R-tree의 이러한 구조적 특징은 GPU의 적용에 여러 문제를

유발한다. 이전 절에서 논의한 GPU와 CPU사이의 메모리 교환은 데이터베이스 시스템에 GPU 적용 시 발생하는 주된 문제점이다. 만약 R-tree와 같은 색인 구조에 GPU를 적용한다면, 노드를 방문할 때 마다 CPU와 GPU사이에 데이터 이동을 필요로 하게 된다. 또한 이것은 GPU에서 한 번에 처리 가능한 데이터의 수를 제한하는 작용을 한다. 예를 들면, R-tree 노드의 m값이 200이고 활용률이 100%인 경우 GPU는 더 처리할 수 있는 쓰레드와 메모리 등의 시스템 자원이 있음에도 불구하고, 한 번에 200개의 데이터 밖에 처리할 수 없다. 또한 이는 앞 절에서 논의한 GPU에서의 메모리 동시 접근 횟수도 감소시킨다. 그리고 이러한 적은 데이터의 처리는 GPU의 global 메모리 영역의 접근 등에 의해서 오히려 CPU보다 오랜 계산 시간을 요구한다. R-tree의 이러한 구조적인 문제는 GPU를 최대한으로 활용할 수 없도록 하며, 이를 해결하기 위한 적절한 기법이 필요하다.

3. 관련 연구

범위 질의의 병렬 처리 기법은 병렬 색인 기법들에서 잘 나타나있다. 기존의 R-tree에 대한 병렬 색인 기법들은 크게 세 가지로 분류할 수 있는데 다수의 디스크와 프로세서를 이용한 기법과 네트워크와 cluster of workstation(COA)를 이용하는 기법 그리고 GPU를 데이터베이스 시스템에 적용한 기법 등이 있다. 다수의 디스크를 이용하는 기법은 데이터를 모든 디스크에 분배하여 질의 수행 시 동시 입출력을 이용하여 성능을 개선하는 기법을 이용한다. 이 기법은 데이터가 어떻게 분포되어 있는가에 따라서 성능의 차이가 크고 또 데이터 분배 시 데이터들 사이의 인접성을 측정하기 위한 계산시간이 필요하다. GPU는 뛰어난 성능으로 인하여 일반 목적으로 이미 많이 연구되고 있다(GPGPU)[1]. 그 중에 한 분야로서 GPU기반의 데이터베이스에 대한 연구가 많이 수행되어 지고 있으나 대부분의 GPU기반 데이터베이스는 aggregation, join등의 연산에 집중되어 있다. 이 절에서는 R-tree와 R-tree를 병렬화하는 각 기법들을 이용하는 병렬 색인 구조에 대하여 소개하고 절의 마지막에서는 GPU를 사용하여 병렬화하는 기법에 대하여 소개한다.

3.1 R-tree

R-tree[3]는 B-tree로부터 확장된 색인 구조로 공간 데이터 및 다차원의 데이터를 색인 및 검색하는데 가장 효율적인 색인 구조이다. R-tree는 공간을 MBR (Minimm Boundary Rectangle)들로 분할하여 저장한다. 상위의 MBR이 하위의 MBR을 포함하는 계층적인 트리 구조를 가지고 있다. 각 노드는 일정 범위 내에서 유동적인 개수의 하위 노드에 대한 정보(MBR, 포인터)를 갖는다. 일반적으로 하나의 노드는 각각 하나의 물리 페이지를 나타내고 페이지의 크기와 노드의 크기 등을 고려하여 적절한 fanout이 설정된다. 이러한 R-tree는 bottom-up 방식으로 형성되는 균형 트리이다. R-tree에서의 검색은 MBR을 기반으로 재귀적인 기법에 의하여 수행되며, 상당한 효율성을 보여준다. 이러한 R-tree의 문제점은 다중 검색 경로에 의한 문제와 노드 활용률 등에 의한 문제를 가지고 있으며, 이러한 문제들을 해결하기 위해 R*-tree 등 많은 연구들이 수행되었다.

3.2 MXR-tree

MXR-tree[4]는 최조의 병렬 처리 공간 색인구조로 다수의 디스크와 프로세서를 사용하여 R-tree를 병렬화한다. MXR-tree는 R-tree를 기반으로 인접한 공간 데이터를 서로 다른 디스크에 분배하여 저장함으로서 입출력의 병렬성을 증가시켜 질의 수행의 효율을 향상 시킨다. 즉, 한 노드에 속한 엔트리들은 공간적으로 서로 인접한 곳에 위치하기 때문에 이 엔트리들을 서로 다른 디스크에 분배함으로써 R-tree를 병렬화한다. 이상적인 노드의 분배를 위해서는 동일 레벨에 속한 모든 노드를 고려하여 가장 인접성이 떨어지는 디스크를 선택해야 하나 이것은 너무 많은 비용을 요구하므로 같은 부모 노드에 속한 엔트리들만을 서로 다른 디스크에 저장한다. 결국 이것은 노드 분배의 불균형을 초래할 수 있다. MXR-tree는 단일 R-tree를 가진 구조이며 모든 노드들은 N개의 디스크에 분배된다. 따라서 MXR-tree는 질의 연산 동안에 N개의 프로세서 각각이 질의 영역과 겹치는 중간 노드를 찾을 때마다 프로세서간의 통신이 필요하며, 통신에 필요한 시간은 사용되는 디스크 수에 비례한다. MXR-tree의 다른 문제점은 노드사이의 인접성을 측정하기위한 계산 시간이 많이 소모되어 구성비용이 크다는 것이다. 또한 R-tree 색인을

기반으로 하기 때문에 다중 경로 문제 등을 가지고 있어서 색인 자체의 효율이 낮다.

3.3 MR-tree and MCR-tree

MR-tree[7]는 하나의 컴퓨터가 서버를 담당하며 N개의 클라이언트와 연결되어진다. 서버는 R-tree의 모든 중간노드를 갖고 있으며 그 최하위 계층에 존재하는 중간노드는 (MBR, site-id, page-id)를 가지고 있다. R-tree의 말단 노드들은 전체 클라이언트들에 분포하며, 서버의 말단 노드는 site-id와 page-id를 통하여 적절한 말단 노드를 찾을 수 있다. 질의의 수행은 다음과 같다. 질의가 도착하면 서버는 질의를 수행하여 필요한 (site-id, page-id)의 리스트를 만든다. 그 후에 해당하는 클라이언트들에게 질의의 MBR과 page의 리스트가 전달되고, 각 클라이언트들은 page를 검색하여 결과를 다시 서버로 전달한다. MCR-tree[8]는 MR-tree와 유사하다. 서버에서는 non-leaf 노드만을 저장하고 있으며, 말단 노드는 (MBR, site-id)로만 이루어져있다. 각 클라이언트는 자신이 가진 데이터에 대하여 독립적인 R-tree를 가지고 있다. MCR-tree에서의 질의 처리는 다음과 같다. 질의처리가 요구되면 서버는 R-tree에서 질의를 수행한다. 수행도중 말단 노드에 도착하면, site-id를 통하여 해당하는 클라이언트에 질의를 전달하는 방식이다.

3.4 In-Memory Grid File

In-Memory Grid File[12]은 Grid File 기법을 GPU에 적용한 기법이다. Grid File과 같은 해싱 기반의 색인 구조는 평면적 구성으로 인해 GPU의 적용이 상대적으로 용이하다. Grid File은 scale과 directory 등을 통하여 전체 데이터를 색인한다. [12]에서 제안하는 기법은 Grid File의 원형은 거의 유지하면서 GPU의 적용 시, GPU의 SIMD 성능을 최대화하여 질의 수행에서의 성능을 개선하였다. [12]에서 제안하는 기법은 다음과 같다. 우선 전체 데이터를 각 차원별로 정렬하고, scale을 구성하기 위해 정렬된 데이터를 샘플링한다. 이렇게 형성된 scale 정보를 토대로, 각 데이터가 저장될 bucket을 찾는다. 그리고 각 bucket의 시작점과 offset을 알아내기 위하여 각 bucket에 속한 엔트리들의 수로 히스토그램을 형성한다. Prefix sum을 통하여 각 bucket의 offset을 구하고 이를 directory에 저장한다. [12]은 Grid File에서 발생할 수 있는, 특정 부분에 데이터가 집중되어 있는 경우에 대하여 고려하였다. 데이터가 특정 부분에 집중되어 있는 경우, GPU의 적용 시 많은 문제를 초래할 수 있다. 왜냐하면, 이것은 결국 특정 메모리 공간에 있는 데이터가 동시에 참조될 가능성이 증가하므로, GPU에 의한 메모리 접근 시, 많은 쓰레드들이 순차적으로 접근하기 때문이다. 이것은 곧 전체 수행 시간의 증가를 의미한다. [12]은 이러한 것을 방지하기 위하여 Skew Handling 기법을 제시하였다. 곧 밀집된 bucket을 여러 하부 bucket으로 분할하도록 한다.

3.5 GPU-CSSTree

이 기법은 CSS-Tree[13]를 GPU에 적용한 기법으로 [14]에서 질의의 동시 처리를 위해 사용된 기법이다. 그림 3에서 볼 수 있듯이, CSS-Tree는 배열을 기반으로 하는 트리로서 GPU의 적용에 유리한 구조를 가지고 있으나 update가 필요한 경우, 전체 tree를 새롭게 구성해야 하는 단점을 가지고 있다. GPU-CSSTree는 이러한 CSS-Tree를 GPU에 적용한 구조로 우선 전체 데이터를 정렬하고 4개의 튜플을 하나의 그룹으로 하여 GPU의 texture 메모리 공간에 저장된다. 이렇게 정렬된 데이터 셋을 말단 노드로 하여 GPU-CSSTree를 구성한다. 기본적으로 배열의 구조를 가지고 있으며, 다른 tree 기반의 인덱스 기법들보다 노드의 활용률이 좋아서 GPU의 적용에 상대적으로 유리한 기법이다. 하지만 이 기법은 CSS-Tree의 문제점도 또한 그대로 가지고 있다. 즉, update에 취약한 구조를 가지고 있어서 update동작

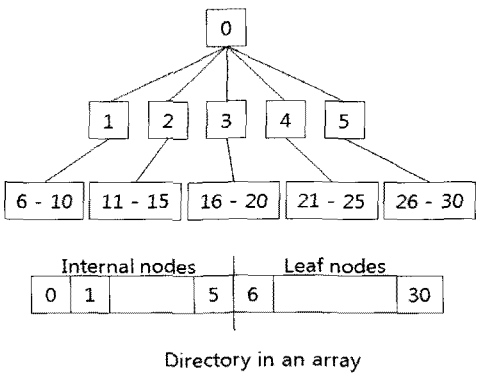

그림 3. CSS-Tree

을 수행하기 위해서 전체 인덱스를 새로 구성해야 하는 문제점을 가지고 있다.

4. 제안 기법

4.1 GPU의 적용

트리와 같은 계층 구조를 가진 색인 구조는 GPU에 적용되기에 불리한 구조를 가지고 있다. 범위 질의 처리에서 실제로 비교되는 MBR들은 각 노드에 분할되어 저장되어 있으며, 서로 연속하지 않는 메모리 공간에 존재하기 때문에 매 노드마다 GPU와의 메모리 교환이 빈번히 이루어져야 하기 때문이다. 또한 이는 GPU가 동시에 접근 가능한 메모리의 용량을 제한시킴으로써, GPU에서의 메모리 동시 접근(coalesced memory access) 횟수를 감소시키고, GPU의 프로세서들의 메모리 접근에서의 오버헤드를 증가시켜, 전체 성능을 저하시키는 원인이 된다. 이러한 오버헤드들을 해결하기 위해서 본 논문에서는 범위 질의를 가속시키기 위해 GPU를 버퍼로 이용하는 기법을 제안한다. 전체 구조는 그림 4와 같다. 실제 데이터를 가지고 있는 R-tree와 말단 노드의 id를 저장하는 해싱 테이블(leaf table), 버퍼링된 영역을 색인하는 메모리 기반의 R-tree(Q-tree), GPU의 global 메모리를 버퍼로 사용하는 버퍼 알고리즘으로 구성된다. R-tree에서의 질의 처리는 디스크 접근 횟수에 비례하여 응답시간이 증가한다. 따라서 일반적인 DBMS에서 제공하는 색인 알고리즘은 버퍼 알고리즘을 포함하고 있다. 그러한 DBMS에서는 노드에 대한 접근 시마다 노드가 버퍼에 존재하는가 여부를 확인하여 디스크 접근 여부를 결정한다. 본 논문은 GPU의 global 메모리 영역에 버퍼를 구성하고 다수의 쓰레드가 동시에 처리하도록 함으로써, 질의의 응답시간을 줄이고자 한다. 또한 단말노드의 엔트리만을 저장하여 질의 처리 요청 시 최대 1회의 버퍼 탐색만을 수행한다. GPU는 메모리 교환에 오버헤드를 가지고 있으며 또한 global 메모리 영역은 크게는 몇 GB에 해당하는 저장 공간을 가지고 있기 때문에 버퍼의 크기가 상당히 커질 수 있다. 따라서 버퍼링된 영역을 따로 저장하여 메모리 교환 및 연산을 최소화할 필요가 있다. Q-tree는 이렇게 버퍼링된 영역을 관리하기 위한 색인 구조이며, 단말노드에 버퍼에서의 시작위치와 버퍼링된 데이터의 양에 대한 정보를 가지고 있다. 질의의 처리가 요구되면 우선 Q-tree와의 교차 영역이 존재하는지 여부에 대한 처리가 수행되며, 교차 영역이 확인되면 버퍼에 대한 처리를 GPU에 요청한다. 이러한 방식으로 메모리 교환을 줄이고 버퍼의 일부분만을 처리하여 연산 량을 줄인다. 이러한 GPU와는 별도로 CPU는 R-tree에서의 질의 처리를 수행한다. R-tree에서의 질의 처리는 기존의 R-tree와 동일하며 단지 단말노드에 대한 디스크 페이지를 읽어오기 전에 leaf table을 확인하여 디스크 접근 여부를 결정한다. leaf table은 버퍼에 저장된 단말노드들의 id를 저장하고 있어서, R-tree에 대한 검색 시, 디스크 접근 여부를 결정한다. GPU_table은 GPU의 global 메모리 영역에 존재하는 버퍼로서 단순히 object들의 배열이다. 배열 형태는 GPU의 data parallel한 특성에 가장 효율적이며, 또한 메모리에 대한 동시 접근을 가장 적절하게 이용할 수 있는 구조이다. GPU_table은 또한 단지 단말 노드만을 버퍼링하고 있다. 중간 노드의 버퍼링은 결국 GPU_table에 대한 중복 접근을 의미하며, 버퍼의 크기가 수 GB에 이를 수 있다는 점과 GPU와의 메모리 교환은 상당한 시간을 필요로 하며 CPU와 동시에 처리될 수 없다는 점에서 비효율적이다. 따라서 본 논문에서 제안하는 버퍼는 단말노드만을 저장하고 있다. 전체 알고리즘은 그림 5와 같다. 질의 요청 시 GPU의 동작 여부를 결정하기 위해서 우선 Q-tree와의 교차 영역을 찾는다. 교차 영역이 확인 되면 GPU는 질의 영역과 GPU_table의 비교 연산을 수행하게 된다. GPU가 GPU_table에 대한 연산을 수행하는 동안, CPU는 R-tree에 대한 탐색을 수행한다. R-tree에 대한 탐색은 기존의 R-tree에서의 검색과 유사하며, 단지 질의가 수행되면서 접근한 말단 노드

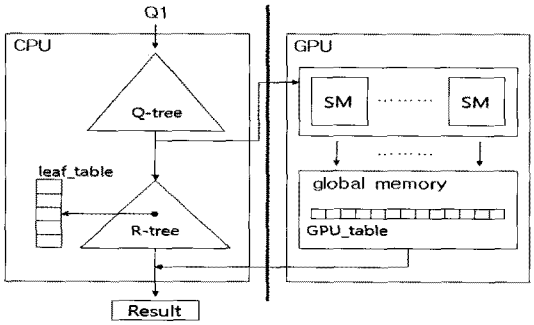

그림 4. 전체 구조

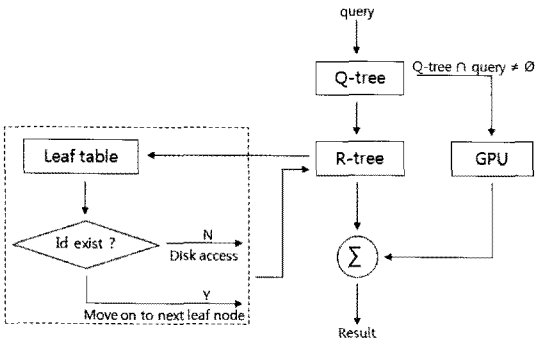

그림 5. 처리 순서도

들은 먼저 GPU_table에 버퍼링된 노드인지 leaf table을 통하여 확인한다. 이미 버퍼링된 노드라면 디스크 접근은 발생하지 않는다. 버퍼링되지 않은 노드라면 버퍼링된 노드들과의 locality를 고려하여 GPU_table에 저장할지 여부를 결정한다. Q-tree에 대한 삽입작업을 통하여 기존에 버퍼링된 노드들과의 locality를 측정할 수 있다. 삽입에 의한 MBR 면적의 증가가 특정 값 T를 초과하는 경우에는 버퍼링하지 않는다. leaf table은 GPU_table에 버퍼링된 단말 노드들의 id를 저장하고 있다. R-tree에서 단말노드에 대한 디스크 접근 여부는 leaf table을 통하여 결정된다. CPU와 GPU에 의하여 각각의 검색이 완료되면 GPU_table에 대한 update작업을 수행한다. update작업은 단말 노드의 버퍼링뿐만 아니라 이미 버퍼링된 노드에 대한 삽입, 삭제 등의 작업으로 인한 변경도 포함된다. update는 Q-tree와 연계되어 수행되어 진다. Q-tree는 GPU_table에 버퍼링되어 있

```
Algorithm for RangeQuery
Begin
1  Start_point = 0;
2  offset = 0;
3  Q_tree_Search(query, start_point, offset );
4  If Q_Rtree is overlap with query range
5     IsOverlap = true;
6     SearchOnGPU( query, start_point, offset );
7  RangeQuery_On_R_tree( query );
8  If IsOverlap is true
9     Get result back from GPU;
10 Update GPU_table;
End
```

는 영역을 색인하기 위한 메모리 기반의 R-tree로서, Q-tree의 노드는 start_index와 count의 두 변수를 추가로 가지고 있다. 질의 수행 시 디스크에서 읽어온 단말 노드의 엔트리들은 GPU에 적재하게 되는데, start_index는 그 첫 위치를 의미하며, count는 질의 수행에 의해 적재된 엔트리들의 총 수를 의미한다. GPU_table은 GPU의 global 메모리 영역에 존재하며, 2 차원 배열의 형태를 가진다. 단말 노드의 엔트리들을 저장하며, GPU에 의해서 병렬로 검색이 이루어진다.

전체 알고리즘은 다음과 같다. 질의의 수행 시, Q-tree에 질의를 먼저 수행하여 버퍼링된 영역들과의 교차영역에 대한 유무를 확인한다(Q_tree_Search). 그림 6에서와 같이, 이전에 버퍼링된 두 영역 A, B와 현재의 질의 영역이 겹치면, 두 노드의 start_index와 count를 이용하여 GPU_table에서의 첫 위치(start_point)와 종료 위치를 알아낸다. start_point는 A노드의 첫 위치를 의미하며, 종료 위치는 B노드의 첫 위치와 count의 합으로 결정된다. 하지만 그림에서와 같이 질의 범위에 포함되지 않지만 검색에 포함되는 엔트리들이 존재하는 단점이 있다. 'SearchOnGPU'는 start_point와 종료 위치를 이용하여 GPU_table을 검색하는 과정이다. 일반적인 GPU에서의 data의 병렬 처리 기법은 그림 7과 같이 고유한 id를 가지고 있는 n개의 쓰레드가 각각 id + K*n(K = 0,1,2,3,...)의 데이터를 처리하도록 한다. 만약 전체 쓰레드의 개수가 256이라면 id가 0인 쓰레드는 0, 256, 512,..번째의 data를 처리하게 된다. 'SearchOnGPU'는 이러한 방식으로 GPU_table을 검색한다. GPU가 GPU_table에 대하여 검색을 수행하는 동안에, R-Tree에서도 질의를 수행한다. 'RangeQuery_On_R_tree'는 일반적인 R-tree의 검색과 유사하다. 질의 수행 시, 단말 노드에 대한 디스크 접근 여부가 leaf table에 의하여

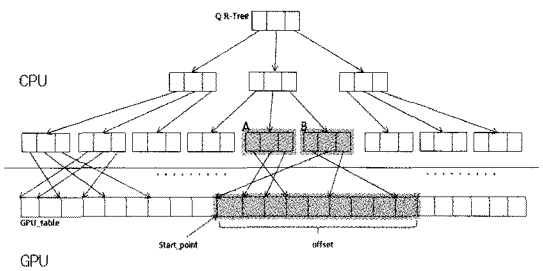

그림 6. 질의 처리

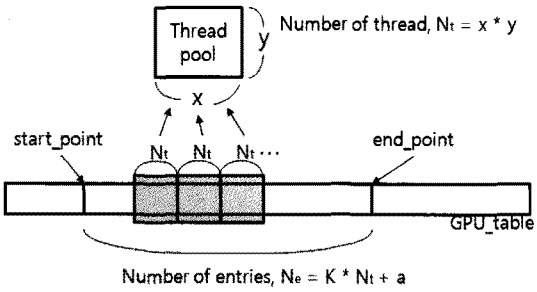

그림 7. GPU search

결정된다. 질의 처리가 완료된 후, update작업을 수행한다. update 작업은 GPU_table에 어떤 변경이 필요한 경우에 대한 작업이며, Q-tree와 연계되어 발생한다.

4.2 Update 전략

버퍼 알고리즘에 의한 성능의 개선은 buffer hit, 즉 요구되는 데이터 중 얼마나 많은 데이터를 버퍼에서 구할 수 있는가에 달려 있다. 따라서 update 전략은 전체 성능에 큰 영향을 미칠 수 있다. GPU의 경우, 메모리 교환에 오버헤드가 존재하기 때문에 버퍼의 크기는 성능에 영향을 미치는 요인 중 하나다. GPU로의 메모리 전송은 그 반대 방향의 메모리 교환보다 빠른 응답시간을 보인다. 본 논문의 실험에서 사용된 GPU의 경우 1.4 GB/s의 대역폭을 가지고 있다. 따라서 Update작업의 잦은 발생은 성능에 큰 영향을 미칠 수 있다.

GPU_table의 update가 필요한 경우는 GPU_table에 data가 추가되는 경우와 GPU_table에 적재된 노드에 add, delete, split, merge 등의 변형이 가해진 경우 발생한다. update는 변형이 가해진 노드 혹은 추가되는 노드의 MBR을 이용하여 Q-tree에서 질의를 수행함으로써 적절한 처리가 가능하다. data 추가의 경우 GPU_table에 충분한 공간이 있다면, 단순히 추가가 가능하지만, 충분한 공간이 없는 경우, Q-tree와 table에서 특정 엔트리와 그에 상응하는 부분에 대한 제거작업이 필요하다. 어떤 부분을 제거할 것인가를 결정하는 것은 수행되는 질의의 분포 등에 의해서 달라질 수 있다. 본 논문에서는 버퍼의 locality를 고려하여 추가될 질의로부터 가장 멀리 위치한 엔트리와 교환하는 방식을 사용하였다. 이것은 GPU_hit를 효율적으로 유지시킬 수 있으나 버퍼에서의 검색 범위를 증가시킬 수 있는 단점이 있다. 노드에 변형이 가해진 경우, 가장 간단한 방법은 실제 변경된 노드를 포함하는 질의를 찾아서, Q-tree, GPU_table 그리고 leaf_table에서 노드를 제거하는 것이다. 그러나 이 기법은 일반적인 R-tree에서와 같이 insert, delete 등의 경우 해당 노드만을 변경하는 것이 아니라, 변경된 노드를 포함하고 있는 Q-tree 노드에 해당하는 전체 영역을 지워야 하기 때문에 비효율적이다. 특히 본 논문에서는 질의의 locality를 기반으로 update 전략을 제안하였기 때문에 제안된 기법은 전체 성능에 큰 영향을 미친다. 본 논문의 경우, 질의의 선택도는 대략 10000개 정도로 제안된 기법은 단 한 노드의 변경으로 이 10000개의 data를 다 GPU_table로부터 삭제하게 된다. 성능 저하의 정도는 결국 변경된 노드의 삭제로 인하여 buffer hit rate가 얼마나 감소할지 여부에 따른다. 만약 10000개의 object가 버퍼에서 삭제되었다면, 50개의 디스크 페이지(fanout 200 기준)가 버퍼에서 삭제된 것이다. 따라서 다음 질의들은 최대 50개의 디스크 페이지를 더 접근해야 한다.

5. 실 험

실험은 Intel(R) Core(TM)2 Quad CPU Q6600과 Nvidia Geforce 8600GT 환경에서, 각각 100,000개의 데이터를 가지고 uniform, skewed, gauss 분포를 가지고 있는 3개의 데이터 셋을 겹침 영역이 적절하게 의도된 질의들을 통하여 진행하였다. GPU의 global 메모리 영역에 존재하는 버퍼의 크기는 10000개의 엔트리를 저장할 수 있도록 정하였다. 객관적인 실험을 위하여, 버퍼가 적용된 R-tree와 비교하여 실험하였으며, 노드의 m값은 200으로 하였다. 버퍼를 통한 이득은 디스크 접근 횟수의 감소를 통하여 얻어지므로, 성능 개선을 위하여 어느 정도 이상의 buffer hit가 필요한지 알아보기 위하여, 변수 GPU hit ratio를 buffer hit 와 전체 결과의 비로 정의한다. GPU hit ratio에 의한 성능 개선 정도를 알아보기 위하여 GPU hit ratio를 증가시키는 반면, selectivity는 상대적으로 일관되게 유지시키면서 수행시간을 측정하였다. 제안 기법의 수행시간은 Q-tree와 R-tree에서의 검색시간 그리고 GPU의 커널함수를 실행시키는 시간을 포함하고 있으며, 질의 수행 후 upload에 소모되

는 시간을 포함한다. 전체 time cost는 다음과 같이 정의된다.

Time cost = CPU time + GPU time + memcpy time

CPU time은 R-tree에서의 검색 시간을 의미하고, GPU time은 Q-tree에서의 검색과 커널함수를 실행시키는데 필요한 시간이며, memcpy time은 결과 셋을 가져오고, update하는 시간을 의미한다. 그림 8은 각 요소가 전체 시간에서 차지하는 비율을 보여주고 있다. 그림 8에서 볼 수 있듯이, CPU time, 즉 R-tree에서의 검색 시간이 가장 큰 부분을 차지하고 있는 것을 볼 수 있었다.

이것은 제안하는 알고리즘에 의한 수행 시간의 감소는 GPU의 적용으로 인하여 감소하는 디스크 접근 횟수와 MBR의 비교 횟수 의하여 결정된다는 것을 말해준다. 즉 GPU hit rate에 의해서 결정된다는 것을 알 수 있다. 다음의 실험은 GPU hit rate에 의하여 수행 시간이 개선되는 정도를 관찰하기 위한 실험이다. 실험은 다음과 같이 진행하였다. 제안하는 기법의 성능 개선의 정도를 관찰하기 위해, 일정한 선택도를 유지하도록 하고 GPU hit rate를 증가시키면서 전체 성능의 개선 정도를 관찰하였다. R-Tree에서 범위 질의를 수행하는 경우와 비교하기 위해서 질의의 수행시간을 측정하였다. 그림 9, 10, 11은 각각 다른 분포를 가지고 있는 데이터 셋을 가지고 수행한 결과이다. 각각의 실험에서 유사한 결과를 확인 할

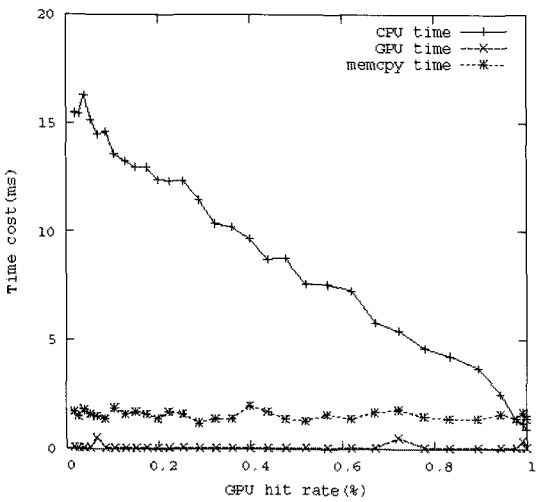

그림 8. Time quota of each element

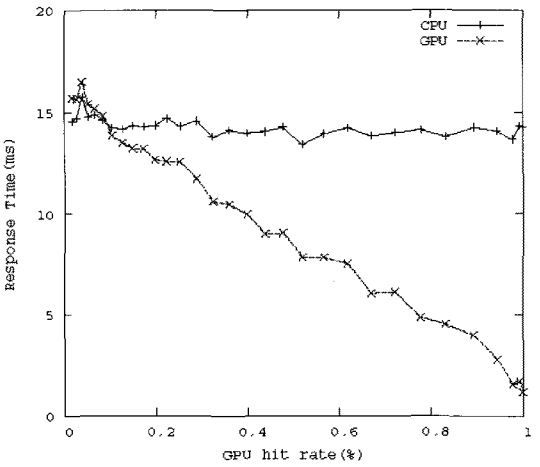

그림 9. 실행 시간 비교(uniform data set)

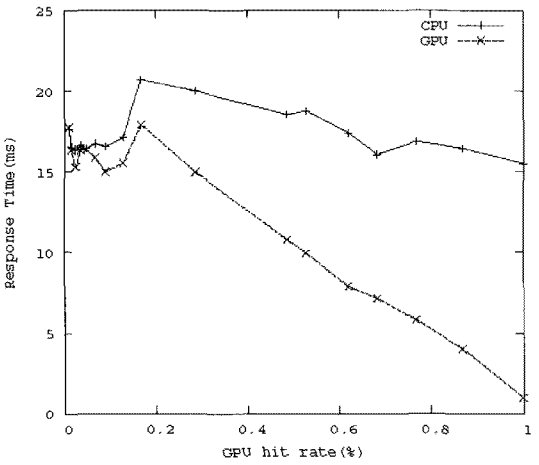

그림 10. 실행 시간 비교(skewed data set)

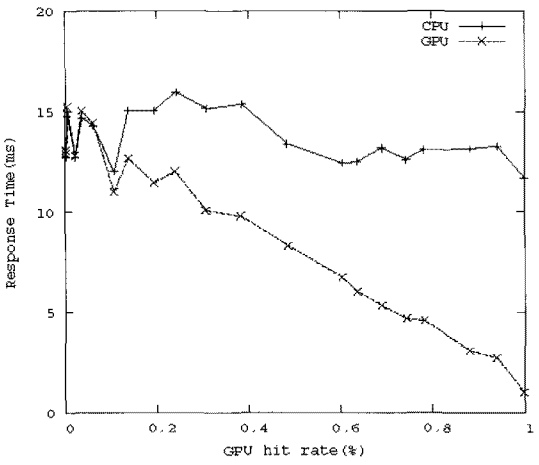

그림 11. 실행 시간 비교(gauss data set)

수 있었다. 첫 질의 수행 시와 GPU rate가 10%보다 낮은 경우에서 단순 R-tree에 의한 질의의 수행 시간이 더 빠른 것을 알 수 있다. 이것은 질의를 수행하면서, leaf table과 말단 노드의 엔트리들을 복사하는 과정에 의하여 오히려 시간을 더 소모하는 것이다. GPU hit rate가 10%정도 되는 시점에서부터 GPU search time이 R-tree time과 거의 유사하게 측정되기 시작하여, GPU hit가 차지하는 비율에 비례하여 수행속도가 증가하는 것을 볼 수 있다. 그림 12는 균일한 분포를 가진 데이터 셋을 통하여, GPU_table의 크기를 전체 데이터 셋의 20%정도로 하고, 선택도를 증가시키면서 응답시간을 측정한 결과이다. 선택도가 증가하면서 전체 수행시간의 차이가 더욱 커지는 것을 알 수 있다. 이것은 선택도가 증가함으로 인하여 GPU hit rate가 증가하기 때문이다.

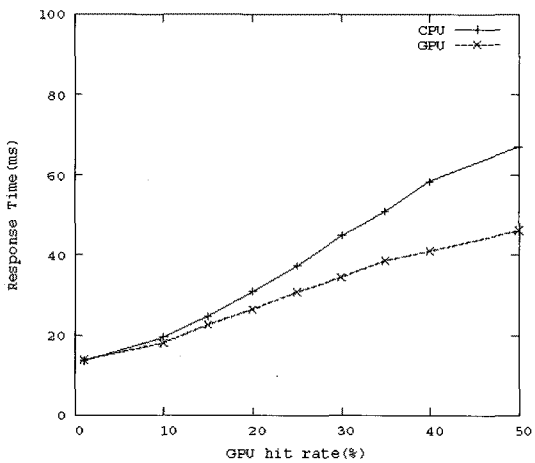

그림 12. Selectivity(uniform data set)

6. 결론 및 향후 연구 과제

현대의 데이터베이스 시스템의 대용량화에 의해서 단일 프로세서와 디스크에 의한 질의의 처리는 시간적인 비용이 크다. 이를 해결하기 위하여 색인 구조의 병렬화에 대한 많은 연구들이 수행되었으며, 그 기법들은 다수의 디스크와 프로세서 그리고 네트워크를 주로 이용한 기법들이다. 이러한 기법들의 단점은 설계비용이 크고 전체 데이터가 디스크 혹은 워크스테이션들에게 어떻게 분배되었는가에 따라서 성능이 결정된다는 것이다. 본 논문에서는 이러한 질의의 병렬처리를 위해서 R-tree에 GPU를 적용하는 기법을 제안하였다.

대부분의 GPU응용에서의 성능 향상은 GPU와 CPU사이의 메모리 교환과, GPU의 메모리 접근에 의해 발생하는 오버헤드의 처리에 따라 결정된다. 기존의 GPU의 트리 기반 색인 구조에의 응용은 레이트레이싱[15], 렌더링[16] 등의 많은 계산량이 요구되는 문제의 해결이 대부분이었기 때문에 이런 문제가 적었으나, 데이터베이스를 위한 R-tree에의 GPU의 적용은 R-tree 자체의 구조 때문에 이러한 문제점들을 발생시켰다. R-tree의 계층적인 구조는 GPU와 CPU사이의 잦은 메모리 교환을 필요로 하고, GPU의 동시메모리 접근 횟수를 저하시킨다. 이러한 문제를 해결하기 위하여본 논문에서는 GPU기반의 버퍼 알고리즘을 제안하였다. 질의에 의하여 방문된 말단 노드들의 엔트리들을 GPU의 메모리에 저장하고 다음 질의 수행 시에 GPU와 CPU에서 동시 검색을 수행함으로써 전체 질의의 수행시간을 감소시키며, 질의 종료 후에 메모리 교환을 수행하여 메모리 교환에 의한 오버헤드를 최소화할 수 있었다. 본 논문에서 제안하는 기법은 버퍼 알고리즘을 사용하기 때문에 전체 성능에 버퍼의 update가 큰 영향을 미친다. 하지만 4장에서 언급된 update기법은 충분하지 않다. R-tree의 기본적인 동작에 의해서 수정된 말단 노드가 GPU_table에 존재할 경우, 해당하는 부분이 R-tree와 일관성을 유지할 수 있도록 수정되어야 한다. 본 논문에서 제안한 update전략은 가장 간단한 전략이나, 메모리 접근을 고려했을 때, 전체 성능을 저하시킬 수 있다. 따라서 이 부분에 대하여 좀 더 효율적인 update 전략에 대한 연구가 필요하다.

참 고 문 헌

[1] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *Proc. Of CGF*, Vol.26, 2007.

[2] *CUDA programming Guide 2.2.*

[3] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching," *Proc. Of SIGMOD84,* 1984.

[4] C.F. Ibrahim Kamel, "Parallel R-Trees," *Proc.*

Of SIGMOD92, 1992.

[5] Xiaodong Fu, Dingxing Wang, Weimin Zheng, and Meiming Sheng, "GPR-Tree: A Global Parallel Index Structure for Multiattribute Declustering on Cluster of Workstations," Proc. Of APDC, pp. 300-306, 1997.

[6] H. Ferhatosmanoglu, A.S. Tosun, G. Canahuate, and A. Ramachandran, "Efficient Parallel Processing of Range Queries Through Replicated Declustering," *Distributed and Parallel Databases,* Vol.20, pp. 117-147, 2006.

[7] N. Koudas, C. Faloutsos, and I. Kamel, "Declustering Spatial Databases on A Multi-Computer Archetecture," Proc. Of EDBT, 1996.

[8] B. Schnitzer and Scott T. Leutenegger, "Master-Client R-Trees: A New Parallel R-Tree Architecture," *Scientific and Statistical Database Management,* pp. 68-77, 1999.

[9] Shuhua Lai, Fenghua Zhu, and Yongqiang Sun, "A Design of Parallel R-tree on Cluster of Workstations," Proc. Of IWDNIS, pp. 119-133, 2000.

[10] Botao Wang, Hiroyuki Horinokuchi, Junihiko Kaneko, and Akifui Makinouchi, "Parallel R-tree Search Algorithm on DSVM. Database Systems for Advanced Applications," Proc. Of SICDSAA, Hsinchu, Taiwan 1999.

[11] J. Nievergelt, H. Hinterberger, and K. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey File Structure," Proc. Of 3rd ECI *Conf.,* Number 123 in LNCS Springer-Verlag, pp. 236-251, 1981.

[12] K. Yang, B. He, R. Fang, M. Lu, N. Govindaraju, Q. Luo, P. Sander, and J. Shi, "In-memory Grid Files on Graphics Proces-Sors," *Proc.* Of the 3rd international workshop on Data management on new hardware, 2007.

[13] Jun Rao and Kenneth A. Ross, "Cache Conscious Indexing for Decision-Support in Main Memory," Proc. Of VLDB, pp.78-89, 1999.

[14] R. Fang, B. He, M. Lu, K. Yang, N. K. Govindaraju, Q. Luo, and P. V. Sander. "Gpuqp: Query Co-processing Using Graphics Processors," Proc. Of the ACM SIGMOD, pp. 1061-1063, 2007.

[15] 장병준, 임인성, "Performance Analysis and Enhancing Techniques of Kd-Tree Traversal Methods on GPU," 한국정보과학회논문지:컴퓨팅의 실제 및 레터, Vol.16, No.2, pp. 177-185, 2010.

[16] 홍인실, 계희원, 신영길, "절단면 재렌더링 기법을 이용한 GPU기반 MIP 볼륨 렌더링," 한국멀티미디어학회논문지, Vol.10, No.3, pp. 316-324, 2007.

유 보 선

2008년 인하대학교 정보공학과 졸업(공학사)
2010년 인하대학교 일반대학원 정보공학과 졸업(공학석사)
2010년~현재 인하대학교 일반대학원 정보공학과 박사재학

관심분야: 데이타 베이스, GPU, 센서네트워크

김 현 덕

2009년 인하대학교 일반대학원 정보공학과 졸업(공학석사)
2009년~현재 인하대학교 일반대학원 정보공학과 박사재학
관심분야: 센서 네트워크, 애드 혹 네트워크, 데이타 베이스

최 원 익

1996년 서울대학교 컴퓨터공학과 졸업(공학사)
1998년 서울대학교 대학원 컴퓨터공학과 졸업(공학석사)
2004년 서울대학교 대학원 전기컴퓨터공학부 졸업(공학박사)

2006년~현재 인하대학교 정보통신공학부 조교수.
관심분야: 모바일/유비쿼터스 컴퓨팅, GIS/LBS, 텔레매틱스

권 동 섭

1998년 서울대학교 컴퓨터공학과 (공학사)
2000년 서울대학교 대학원 컴퓨터공학과(공학석사)
2005년 서울대학교 대학원 컴퓨터공학과(공학박사)

2005년~2006년 삼성전자 SW연구소 책임 연구원
2006년~명지대학교 컴퓨터공학과 부교수
관심분야: 데이터베이스, 질의 처리, 전자상거래