CUDA 프레임워크 상에서 스카이라인 질의처리 알고리즘 최적화

(Optimizing Skyline Query Processing Algorithms on CUDA Framework)

민 준 † 한 환 수 †† 이 상 원 †††
(Jun Min) (Hwan Soo Han) (Sang-Won Lee)

요 약 GPU는 대용량 데이터 처리를 위해 특화된 멀티 코어 기반의 스트림 프로세서로서 빠른 데이터 처리 속도 및 높은 메모리 대역 등의 장점을 가지며, CPU에 비해 가격이 저렴하다. 최근 이러한 GPU의 특성을 활용하여 범용 컴퓨팅 분야에 활용하고자 하는 시도가 계속되고 있다. 엔비디아에서 발표한 범용 병렬 컴퓨팅 아키텍처인 쿠다(CUDA) 프로그래밍 모델의 경우 프로그래머가 GPU 상에서 동작하는 범용 어플리케이션을 보다 손쉽게 개발할 수 있도록 지원한다. 본 논문에서는 쿠다 프로그래밍 모델을 이용하여 기본적인 중첩-반복 스카이라인 알고리즘을 병렬화시킨다. 그리고 스카이라인 알고리즘의 특성을 고려하여 GPU 자원을 효율적으로 사용할 수 있도록 GPU의 메모리 및 명령어 처리율에 중점을 두고 단계적인 최적화를 진행한다. 최적화 단계에 따라 각각 다른 성능 개선이 나타나는 것을 확인하였으며, 그 결과 기본 병렬 중첩-반복 알고리즘에 비해 평균 80%의 성능이 향상됨을 확인하였다.

키워드 : 범용 그래픽 처리 장치, 쿠다, 스카이라인, 스트림 프로세서, 대용량 데이터 처리

Abstract GPUs are stream processors based on multi-cores, which can process large data with a high speed and a large memory bandwidth. Furthermore, GPUs are less expensive than multi-core CPUs. Recently, usage of GPUs in general purpose computing has been wide spread. The CUDA architecture from Nvidia is one of efforts to help developers use GPUs in their application domains. In this paper, we propose techniques to parallelize a skyline algorithm which uses a simple nested loop structure. In order to employ the CUDA programming model, we apply our optimization techniques to make our skyline algorithm fit into the performance restrictions of the CUDA architecture. According to our experimental results, we improve the original skyline algorithm by 80% with our optimization techniques.

Key words : GPGPU, CUDA, Skyline, Stream Processor, Large Size data Processing

· 본 연구는 지식경제부 및 정보통신산업진흥원의 대학 IT연구센터 지원사업(NIPA-2010-(C1090-1021-0008)), 서울시의 지원으로 수행한 '서울시 산학연 협력사업(PA090903)', 교육과학기술부 재원으로 한국연구재단의 지원을 받아 실시한 '중견연구자 지원사업(NRF 2009-0084870)'과 '기초연구사업(NRF 2010-0025649)'의 연구결과입니다.

† 학생회원 : 성균관대학교 임베디드 소프트웨어학과
hornetmj@hanmail.net
†† 종신회원 : 성균관대학교 정보통신공학부 교수
hhan@skku.edu
††† 정 회 원 : 성균관대학교 정보통신공학부 교수
wonlee@ece.skku.ac.kr

논문접수 : 2010년 8월 23일
심사완료 : 2010년 9월 10일

정보과학회논문지: 데이타베이스 제37권 제5호(2010.10)

1. 서 론

최근 GPU(Graphics Processing Unit)의 연산 성능은 눈부시게 발전해 왔다[1]. GPU는 병렬적으로 대량의 데이터를 처리하기 위해 특화된 다수의 코어(core)를 탑재한 보조 프로세서(co-processor)로서 CPU에 비해 가격이 저렴하고 데이터 처리 속도와 메모리 대역 등이 탁월하다는 특징을 가진다[1]. 최근 이러한 GPU 처리 능력을 컴퓨터 그래픽 분야가 아닌 범용 컴퓨팅 분야에 활용하고자 하는 시도가 계속 되고 있다[2-6]. 이처럼 전통적으로 CPU가 담당하던 연산을 GPU의 병렬 연산 처리 능력을 활용하여 처리하는 기술을 GPGPU(General-Purpose computing on GPU)라고 한다[6]. GPGPU의 사용 목적은 CPU의 제한된 병렬성[7-9]을 극복하고 대용량 데이터 및 복잡한 알고리즘을 보다 효율적이고 빠

르게 처리하는 것이다[1].

엔비디아(NVIDIA)와 에이티아이(ATI) 등의 회사들은 고성능 컴퓨팅용 GPU를 선보이고 있으며, 특히 엔비디아의 테슬라(Tesla)와 같은 GPU는 개인용 슈퍼컴퓨터라는 새로운 지평을 열어가고 있다. 이와 같은 CPU와 GPU의 성장 추세는 개인용 컴퓨터(PC)에서도 멀티 코어의 성능을 최대화하기 위한 병렬 처리 프로그래밍의 중요성을 더욱 커지게 만들고 있다. 엔비디아는 최근 쿠다(CUDA, Compute Unified Device Architecture)라는 범용 병렬 컴퓨팅 아키텍처를 발표하였다[1,10,11]. 쿠다는 기존의 셰이딩(shading) 언어[12] 기반의 범용 GPU 프로그래밍과 달리 손쉽게 스레드 레벨 병렬 처리를 제어할 수 있다는 점에서 프로그래머의 자유도가 획기적으로 높아졌으며, 대부분의 병렬 알고리즘을 쿠다 기반으로 구현할 수 있게 되었다.

본 논문에서는 엔비디아의 쿠다 병렬 프로그래밍 모델을 이용하여 가장 기본적인 중첩-반복(nested-loops) 스카이라인 알고리즘을 병렬화시키고, GPU 상에서 성능 극대화를 위한 단계적인 최적화 방안들을 제시한다. 일반적으로, 영상 처리 및 컴퓨터 비전(computer vision) 응용과 같이 병렬화 적용을 위한 데이터 구조 및 알고리즘이 단순한 분야가 GPU 처리 구조에 적합한 것으로 알려져 있는데[13,14], 스카이라인 처리와 같이 상대적으로 복잡한 알고리즘의 경우에도 해당 알고리즘의 특성과 최신 GPU의 기능을 활용하여 성능을 향상시킬 수 있음을 보이겠다.

2. 배경 지식 및 관련 연구

2.1 GPU 구조

GPU는 스트림 프로세서(stream processor)로서 순차적으로 하나의 명령을 수행하고 메모리를 수정하는 직렬 프로세서(serial processor, CPU)와 달리 대량의 입력 데이터 집합에 대하여 하나의 명령을 병렬적으로 처리하고 출력 데이터 집합을 동시에 발생시킨다[6,11]. 각 스트림 프로세서로 전달되는 데이터들은 기본적으로 다른 데이터와의 종속성 없이 독립적으로 처리된다[10,11]. 그림 1은 엔비디아 지포스(GeForce) 계열 그래픽 카드 구조를 간략히 도식화하여 나타낸 것이다. 디바이스 메모리(device memory)에 스트리밍 멀티프로세서(Streaming Multiprocessor, SM)들이 연결되어 있고, 각 멀티프로세서에는 여러 개의 스트림 프로세서(Stream Processor, SP)가 탑재되어 있다. 같은 멀티프로세서에 탑재된 스트림 프로세서들은 공유 메모리(shared memory), 상수 캐시(constant cache), 그리고 텍스처 캐시(texture cache)를 공유하여 사용한다. CPU와 비교하면 GPU는 제어 부분이 거의 없고 연산기(ALU)가 대부분을 차지하는 구조로서 분기가 없으며 연산이 많은 프로그램의 실행 시 탁월한 성능을 보여준다[10].

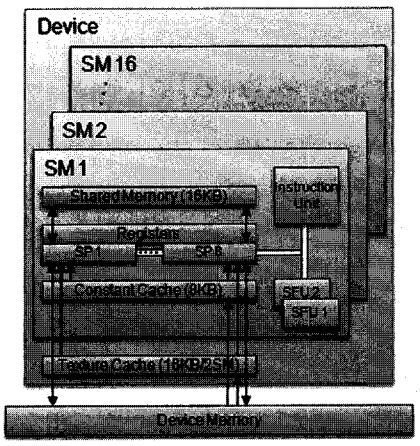


그림 1 GPU 구조

2.2 쿠다 병렬 프로그래밍 모델

쿠다 병렬 프로그래밍 모델은 C언어를 확장한 형태로 이루어져 있으며, CPU 환경에서 처리되는 호스트 코드(host code)와 GPU 환경에서 처리되는 디바이스 코드(device code)로 구성된다[10]. 특히 GPU에서 처리되는 함수를 커널(kernel)이라고 하며, 일반적인 프로그램 함수와 달리 GPU 상에 생성된 n개의 스레드들에 의해 총 n번 실행되어 진다[10]. 이처럼 쿠다 프로그래밍 모델은 스레드들이 동일한 커널을 여러 번 수행하는 SIMT (Single-Instruction, Multiple-Thread) 방식을 취한다[10,11]. 또한 많은 수의 코어와 다수의 스레드를 효과적으로 활용하기 위한 스레드 관리 기법, 공용 메모리 처리, 스레드 동기화 등 병렬 프로그래밍을 위한 기본 아키텍처를 지원한다[11].

그림 2는 쿠다 스레드 구조를 나타낸다. 가장 기본 실행 단위로서 연산을 담당하는 스레드, 스레드들로 구성된 블록(block), 그리고 여러 블록들로 구성된 그리드(grid)로 이루어지며, 그리드는 최대 2차원 블록으로 구성될 수 있다. 또한 한 블록의 스레드 구성은 3차원까지 허용되며, 최대 512개의 스레드들을 포함할 수 있다. 블록은 멀티프로세서 스케줄링(scheduling)의 기본 단위로서 처음 할당된 멀티프로세서에서 실행이 완료될 때까지 계속 수행되며, 블록 간 실행 순서 및 처리는 상호 독립적이다. 또한 실행 준비된 다른 블록들이 사용할 충분한 공유 자원이 남아 있다면, 동시에 여러 개의 블록들이 할당되어 실행될 수 있다. 각 멀티프로세서에 할당된 블록은 연속적인 32개 스레드 집합인 워프(warp)로 나뉘어 독립적으로 실행되며, 워프 스케줄러(warp scheduler)에 의해 스케줄링 된다. 워프에 속한 스레드들은 동시에 하나의 명령어만을 실행시키기 때문에 워프의 모든 스레드들이 같은 실행 경로를 가질 때 최대

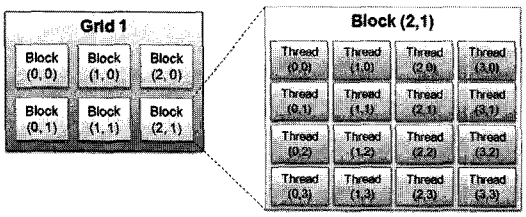


그림 2 쿠다 스레드 구조

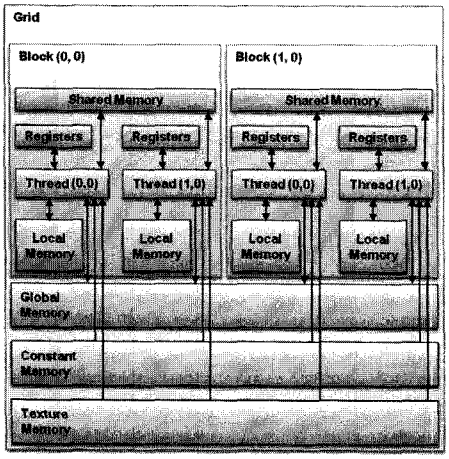


그림 3 쿠다 메모리 구조

효율을 얻을 수 있다. 워프를 다시 연속적인 16개 스레드 집합으로 나눈 것을 하프-워프(half-warp)라고 한다.

GPU는 스레드들의 효율적인 데이터 처리를 위해 다양한 종류의 메모리를 지원하며, 이는 크게 온칩(on-chip)과 오프칩(off-chip) 메모리로 구분된다[11]. 그림 3은 GPU의 메모리 구조와 각 메모리 간의 연관 관계를 도식화한 것이다. 각 스레드는 실행되는 동안 개별적인 레지스터(register)와 지역 메모리(local memory) 공간을 할당 받아 사용하며, 같은 블록 안의 스레드들은 공유 메모리(shared memory)를 통해 서로 통신할 수 있다. 또한 생성된 모든 스레드들은 전역 메모리(global memory)의 임의의 주소 공간에 접근하여 쓰거나 읽을 수 있다. 레지스터와 공유 메모리는 온칩 메모리로서 오프칩 메모리인 지역 메모리와 전역 메모리에 비해 훨씬 빠른 접근 시간(access time)과 접근 비용(overhead)을 가지며, 공유 메모리는 전역 메모리의 캐시(cache)처럼 사용된다. 따라서 공유 메모리를 얼마나 효과적으로 사용하는가에 따라 프로그램의 전체 수행 성능에 큰 영향을 미칠 수 있다.

2.3 스카이라인 질의

스카이라인 질의란 전체 객체 집합으로부터 질의자의 선호도에 따라 관심을 가질만한 객체들을 뽑아내는 연산이다[15-20]. 일반적으로, n개의 속성을 가지는 객체들의 전체 집합을 DS라하고, 이에 속하는 객체 p, q가 있다고 할 때(p, $q \in DS$), 객체 p가 객체 q와 비교하여 최소 하나의 속성이 질의자의 선호도를 더욱 만족시키고, 나머지 n-1개의 속성들이 상대적으로 불만족스럽지 않은 경우 객체 p를 질의자가 관심을 가질만한 객체라고 한다[15]. 또한 전체 객체 집합 DS 중 이런 객체들의 모임을 스카이라인 집합 SKY라 하며($SKY \subseteq DS$), 질의자는 스카이라인 집합에서 최종적으로 원하는 정보(객체)를 선택할 수 있다.

예를 들어, 질의자가 휴양지의 호텔 예약을 위해 '숙박 요금은 저렴하고, 해변과의 거리는 가까운 호텔'에 관한 정보를 원한다고 가정하자. 일반적으로, 해변과 가까운 호텔 일수록 가격이 비싼 반비례 관계를 형성하므로 조건에 일치하는 최적의 호텔 하나를 결정해 줄 수는 없다. 따라서 질의자가 관심을 가질만한 호텔 정보를 제공해주고, 최종 선택은 질의자 선호도에 의해 결정될 수 있도록 해야 한다.

그림 4는 호텔 스카이라인 집합과 객체 H_8을 기준으로 다른 객체들과의 지배 관계[15]를 나타낸 것이다. H_8을 원점으로 하는 직각좌표계에서 1사분면에 위치한 객체들은 H_8이 지배하는 피지배 객체(dominatee) 집합이고, 3사분면에 위치하는 객체들은 H_8을 지배하는 지배객체(dominator) 집합이다. 또한 2, 4분면에 위치하는 객체들은 H_8과 비교할 수 없는 객체(incomparable)로서 H_8이 지배하지도 H_8을 지배하지도 않는 객체들이다. 전체 객체 집합 중에 주워진 조건을 만족시키는 스카이라인 집합을 구하는 기본 원리는 객체들 간의 지배 관계를 판별(dominance test)하여 피지배 객체들을 제거하는 것이다[15]. 그림 4에서 실선으로 연결된 객체 집합 {H_1, H_4, H_7, H_{11}, H_{12}, H_{13}}은 저렴한 호텔 숙박 요금과 근접한 해변과의 거리 두 가지 속성에 의해 결정된 스카이라인 집합을 나타낸다.

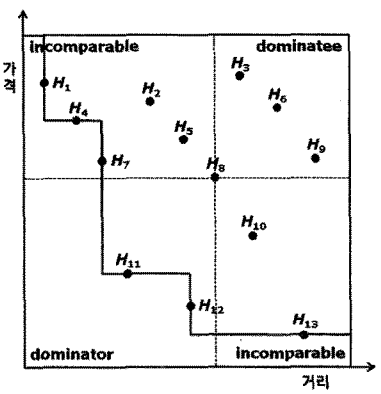


그림 4 스카이라인 집합과 객체 지배 관계

2.4 병렬 스카이라인 처리[21]

스카이라인 연산이 소개된 이후 중첩-반복(nested-loops), 분할 정복(divide-and-conquer), 그리고 인덱스 등에 기반한 다양한 스카이라인 알고리즘이 소개되었다[15-20]. 특히 최근에 발표된 영역 트리(R-tree) 인덱스를 이용하는 BBS(Branch-and-Bound) 알고리즘[20]은 싱글 코어 구조에서 빠르게 동작하는 알고리즘 중 하나이며, 영역 트리를 입력으로 받아 스카이라인 집합을 구하는 것이 기본 동작 원리이다. Parallel BBS 알고리즘[16]은 BBS 알고리즘에서 가장 많은 계산 시간을 소모하는 객체 간 지배 관계 판별 부분을 멀티 코어 환경에 맞게 병렬화한 것이다.

pskyline 알고리즘[16] 또한 멀티 코어 환경에서 스카이라인 집합을 구하는 알고리즘으로 맵(map)과 리듀스(reduce) 단계를 가진다. 맵 단계는 전체 객체 집합을 코어의 수(n개)에 맞게 부분적 객체 집합 DS_1, DS_2, …, DS_n으로 등분하여 코어 별로 부분적 스카이라인 집합 SKY_1, SKY_2, …, SKY_n을 구하는 과정을 말한다. 리듀스 단계는 부분적 스카이라인 집합 SKY_i(임의의 스카이라인 부분 집합)의 객체들을 각 코어에 분할 할당하여 SKY_j(임의의 스카이라인 부분 집합)의 객체들과 병렬적인 지배 관계 판별을 통해 합병된 하나의 스카이라인 부분 집합 $SKY'(SKY_i \text{ ++ } SKY_j)$을 구하는 과정을 반복적으로 수행하여 전체 스카이라인 집합 SKY를 구하는 과정을 말한다. 또한 참고문헌[22]에 기반하여 SIMD 벡터 연산을 이용한 병렬 스카이라인 처리 역시 제안되어질 것으로 예상된다[22,23].

멀티 코어 CPU 상에서의 병렬 스카이라인 처리는 전체 객체 집합을 코어의 수에 기준하여 적절히 분할하고 이를 독립적으로 처리한 후 합병한다[24]. 반면, GPU 상에서 코어의 수에 기준하여 전체 객체 집합을 분할하고 이를 개별 처리할 경우 동시에 같은 명령어를 수행하는 워프의 스레드들이 서로 다른 실행 경로를 가지게 되고, 이는 순차적으로 처리되므로 매우 비효율적이다. 또한 하드웨어적으로 관리되는 CPU 상의 캐시 메모리와 달리 GPU의 경우 프로그래머에 의해 소프트웨어적으로 공유 메모리가 사용되어지기 때문에 한 블록의 스레드들이 모두 독립적으로 서로 다른 부분 객체 집합에 대한 스카이라인 처리를 수행할 경우 메모리 대역을 낭비하게 된다. 본 논문에서는 이러한 GPU와 스카이라인 알고리즘 고유의 특성을 적절히 조화하여 GPU 자원을 효율적으로 활용하고 전체 프로그램 성능을 향상시킬 수 있는 방법을 제안한다.

3. 병렬 중첩-반복 스카이라인 알고리즘

일반적으로, GPU 처리 구조는 영상 처리와 컴퓨터 비전 응용과 같이 병렬화를 위한 데이터 구조 및 알고리즘이 단순한 분야에 적합한 것으로 알려져 있다[13,14]. 본 장에서는 스카이라인 처리와 같이 상대적으로 복잡한 알고리즘의 경우에도 해당 알고리즘의 특성과 최신 GPU의 기능을 활용하여 성능을 향상시킬 수 있음을 보이겠다.

엔비디아의 쿠다 병렬 프로그래밍 모델을 이용해 작성된 프로그램이 최적의 성능을 얻기 위해서는 기본적으로 최대한 많은 부분을 병렬 처리해야 한다[11]. 또한 메모리 처리율(memory throughput)과 명령어 처리율(instruction throughput)이 최대가 될 수 있도록 최적화시켜야 한다[10,11]. 메모리 처리율을 높이기 위한 가장 기본적인 전략은 공유 메모리를 효율적으로 사용하여 전역 메모리에서 읽어오는 데이터를 최소화 시키는 것이다[10]. 한 블록 안의 스레드들은 같은 공유 메모리 주소를 사용하기 때문에 사용해야할 데이터들이 이미 같은 블록의 다른 스레드들에 의해 공유 메모리로 읽혀진 경우 전역 메모리에 대한 접근 없이 바로 사용할 수 있다. 이처럼 공유 메모리를 전역 메모리의 캐시처럼 사용하고 같은 블록의 스레드들에 의한 데이터 재사용률을 높임으로서 전역 메모리에 대한 접근을 줄이고 전체 성능을 향상 시킬 수 있다. 명령어 처리율은 프로그램의 수행 흐름을 제어하는 명령어(*if, switch, do, for, while*)에 의해 상당한 영향을 받는다[10]. 한 워프에 속한 스레드들이 흐름 제어 명령어에 의해 서로 다른 실행 경로를 취하는 경우 병렬적으로 처리되지 못하고 각 실행 경로를 순차적으로 수행하게 된다. 이를 분기하는 워프(divergent warp)가 발생했다고 말하며, 이 경우 GPU에서 지원하는 하드웨어 파워를 낭비하게 된다.

본 논문에서는 이러한 GPU의 특성을 활용하여 가장 기본적인 중첩-반복(nested-loops) 스카이라인 알고리즘[15]을 병렬화시키고, 성능 향상을 위해 알고리즘 고유의 특성을 GPU 환경에 맞게 단계적으로 수정하는 방법을 제안한다. 중첩-반복 알고리즘은 가장 기본적인 스카이라인 처리 방법으로 전체 객체들을 서로 한 번씩 비교하여 주어진 조건에 부합하는 스카이라인 집합을 구한다. 이는 매우 비효율적인 방법으로 병렬 처리를 통해 상당한 성능 향상을 얻을 수 있다. 중첩-반복 알고리즘을 병렬화시키는 기본 개념은 다수의 스레드를 생성하고 각 스레드는 한 객체와 다른 객체들 사이에 발생하는 비교 연산을 독립적으로 수행하는 것이다. 이는 블록 간 상호 독립적으로 스케줄 되어 실행되는 쿠다 프로그래밍 모델의 특성을 매우 효과적으로 활용할 수 있다. 그림 5는 n개 객체 집합을 k개의 스레드가 병렬 처리하는 중첩-반복 알고리즘을 간략하게 도식화한 것이다. 생성된 스레드의 수가 전체 객체 집합의 수보다

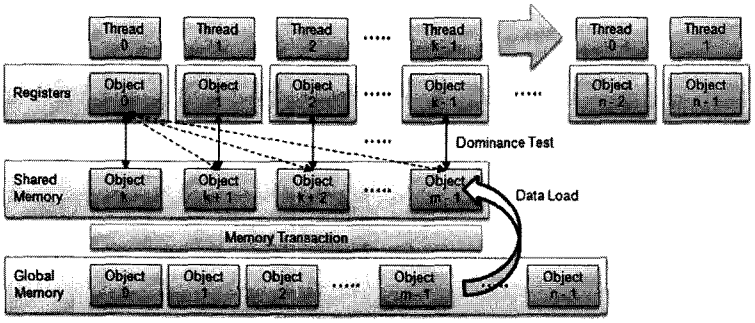


그림 5 병렬 중첩-반복 알고리즘

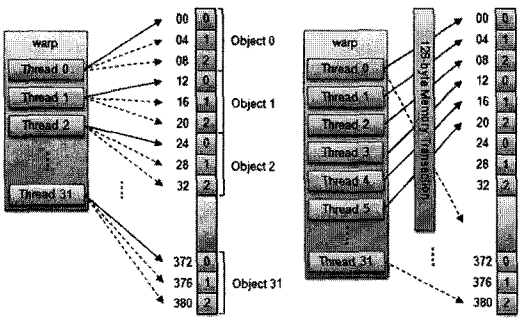
많거나 같은 경우($n < k$, $n = k$) 기본 병렬 알고리즘에 의해 잘 처리되지만, 스레드에 의해 사용되는 레지스터 수 및 블록이 사용할 수 있는 공유 메모리 크기 등 하드웨어적 제약에 의해 생성할 수 있는 스레드 수는 제한된다[11]. 따라서 최적의 성능을 얻기 위해서는 하드웨어 제약과 처리하는 데이터 특성을 고려하여 적합한 개수의 스레드를 생성하고, 처리해야할 객체의 수가 생성된 스레드 수보다 많은 경우($n > k$) 그림 5의 스레드 0, 스레드 1과 같이 기본 병렬 알고리즘을 반복적으로 수행하여 주어진 조건에 부합하는 스카이라인 집합을 얻을 수 있다. 병렬 중첩-반복 스카이라인 알고리즘을 메모리 처리율과 명령어 처리율 관점에서 최적화시키고 성능 향상 정도를 비교해보자.

4. 최적화

4.1 전역 메모리 접근 패턴 변경

쿠다 프로그래밍 모델에서 전역 메모리는 32-, 64-, 또는 128-바이트 크기의 메모리 트랜잭션에 의해 접근되어진다[10]. 메모리 트랜잭션은 기본적으로 그 크기에 정렬(align)되어 발생하기 때문에 같은 크기로 정렬된 32-, 64-, 또는 128-바이트 세그먼트(segment) 단위로 데이터를 읽거나 쓸 수 있다[10]. 워프가 전역 메모리에 접근하는 명령어를 실행하는 경우 이에 속한 32개의 스레드들이 접근하는 데이터 타입과 메모리 주소의 분산 정도에 따라 하드웨어적으로 하나 또는 그 이상의 메모리 트랜잭션을 발생시켜 협력적으로 전역 메모리에 대한 접근을 처리한다[10]. 일반적으로, 같은 명령어 처리에 대해 더 많은 트랜잭션이 필요할수록 스레드들에 의해 사용되지 않는 불필요한 워드(word)의 전송이 발생하여 명령어 처리율을 감소시키는 원인이 된다[10,11]. 예를 들어, 워프의 32개 스레드들이 각각 연속된 4-바이트 크기의 단정밀도 부동 소수점(single precision floating point) 형식을 읽을 경우 128-바이트 크기의 메모리 트랜잭션을 발생시켜 한 번에 읽어 올 수 있지만, 스레드들이 서로 다른 형식의 데이터에 접근하거나 주소 범위를 벗어나는 데이터에 접근하는 경우 한 번 이상의 메모리 트랜잭션이 발생하게 된다.

스카이라인 질의에서 사용되는 기본 데이터 단위는 2개 이상의 속성으로 구성된 객체이다. 병렬 중첩-반복 알고리즘에서 사용되는 객체 집합은 전역 메모리 상에 순차적으로 나열되어 있으며, 설명의 편의를 위해 첫 번째 객체는 0번지에 위치하고 각 속성은 4-바이트 크기의 단정밀도 부동 소수점 형식이라고 가정한다. 그림 6은 워프의 스레드들이 객체들을 공유 메모리로 읽어가기 위해 전역 메모리에 접근하는 방법을 간략하게 도식화한 것이다. 객체 하나가 3개의 속성으로 구성되어 있기 때문에 워프의 스레드들은 총 384-바이트 범위의 전역 메모리에 접근한다. 그림 6의 (a)처럼 스레드들이 객체 단위로 메모리에 접근하는 경우 12-바이트 크기의 객체를 한 번에 읽을 수 없기 때문에 내부적으로는 첫 번째 속성부터 순차적으로 읽어 들인다. 이 경우 요청된 데이터 전송을 완료하기 위해서는 총 9번의 128-바이트 메모리 트랜잭션이 발생하게 된다. 그림 6의 (b)방법은 이러한 문제점을 개선한 것이다. 워프의 스레드들은 객체의 속성 단위로 메모리에 접근하고, 3번(속성 수) 반복하여 순차적으로 데이터를 전송한다. 이 경우 총 3번



(a) 일반적 메모리 접근 (b) 개선된 메모리 접근

그림 6 전역 메모리 접근 패턴 변경

의 128-바이트 메모리 트랜잭션에 의해 데이터 전송이 완료되기 때문에 상대적으로 메모리 대역폭 낭비를 줄일 수 있다.

4.2 흐름 제어 명령어 제거

앞서 언급한 것과 같이 GPU는 스트림 프로세서로서 제어 부분이 거의 없고 연산기(ALU)가 대부분을 차지하는 구조이다[6,11]. 또한 쿠다 프로그래밍 모델에서 생성된 모든 스레드들은 같은 프로그램을 수행하며, 워프 단위로 스케줄 되어 동시에 하나의 명령어만 처리하는 SIMT 구조를 취한다[10,11]. 따라서 프로그램 수행 중 분기 발생이 적고 상대적으로 처리해야할 연산이 많은 경우 매우 효율적이다.

그림 7은 스카이라인 알고리즘에서 가장 핵심이 되는 부분을 나타낸다. 알고리즘을 간략히 설명하면, 두 객체의 첫 번째 속성부터 마지막 속성까지 순차적으로 비교하여 두 객체 사이의 지배 관계를 판별하는 것이다. 예를 들어, n개의 속성으로 구성된 두 객체 p, q 간의 비교에서 m-1($n > m$+1)번째 속성까지 객체 p가 객체 q를 지배하고 m번째 속성 간의 비교에서도 객체 p가 객체 q를 지배하면, m+1번째 속성 간의 비교를 계속 수행한다. 반면, m번째 속성 간의 비교에서 객체 p가 객체 q를 지배하지 않으면 두 객체는 상호 비교할 수 없는 지배 관계로 판별되고 속성 간의 비교는 중단된다. 이는 일반적인 CPU 컴퓨팅 환경에서 단일 스레드가 두 객체 간의 비교 시 불필요한 연산을 수행하지 않음으로서 처리 효율을 높이기 위함이다.

```
DOMINANCE_TEST (object p, object q) {

  dominator = DOM_UNKNOWN;

  for(i=0; i<n; i++) {
    if (p.elem[i] < q.elem[i]) {
      if (dominator == DOM_RIGHT) {
        dominator = DOM_UNKNOWN;
        break;
      }
      dominator = DOM_LEFT;
    }
    else if (p.elem[i] > q.elem[i]) {
      if (dominator == DOM_LEFT) {
        dominator = DOM_UNKNOWN;
        break;
      }
      dominator = DOM_RIGHT;
    }
  }
}
```

그림 7 지배 관계 판별 알고리즘

```
DOMINANCE_TEST (object p, object q) {

  dominate_left = initialize to 1;
  dominate_right = initialize to 1;
  all_equal = initialize to 1;

  for(i=0; i<n; i++) {
    column_le = (p.elem[i] LE q.elem[i]);
    column_ge = (p.elem[i] GE q.elem[i]);
    column_eq = (p.elem[i] EQ q.elem[i]);

    dominate_left = (dominate_left AND column_ge);
    dominate_right = (dominate_right AND column_le);
    all_equal = (all_equal AND column_eq);
  }

  dominate_left = (dominate_left XOR all_eq);
  dominate_right = (dominate_right XOR all_eq);
}
```

그림 8 개선된 지배 관계 판별 알고리즘

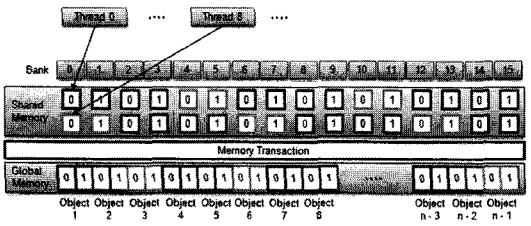
쿠다 프로그래밍 모델에서 스레드들은 SIMT 방식으로 동작하기 때문에 두 객체의 속성 간의 비교 결과에 따라 서로 다른 실행 경로로 분기하는 그림 7의 알고리즘 수행 시 분기하는 워프가 발생한다. 이 경우 발생한 모든 실행 경로를 순차적으로 처리하게 되며, 스트림 프로세서인 GPU의 특성을 효율적으로 활용할 수 없다. 그림 8은 쿠다 프로그래밍 모델의 명령어 처리율을 극대화시키기 위해서 흐름 제어 명령어인 *if, else if, break* 구문을 제거하고, 논리 연산자(*and, or, xor*)를 활용하여 스카이라인 여부를 판단하도록 수정한 것이다. 따라서 수행 중인 워프의 스레드들은 서로 다른 실행 경로로의 분기 없이 같은 명령어를 실행하기 때문에 병렬적으로 모든 처리를 수행한다. 반면, 그림 7의 판별 알고리즘의 경우 최소 2개의 속성 비교를 통해 객체 간의 지배 관계가 결정된 경우 추가적인 속성 비교 수행 없이 *break* 구문에 의해 판별 알고리즘을 종료하지만, 그림 8의 경우 불필요한 나머지 속성 간의 비교를 끝까지 수행해야 하는 단점이 발생한다. 그렇지만 결과적으로 GPU의 명령어 처리율을 극대화함으로서 전체 프로그램 성능을 대폭 향상시킨다.

4.3 공유 메모리 뱅크 충돌 제거

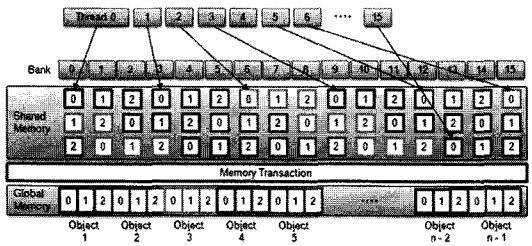
공유 메모리는 온칩 메모리로서 전역 메모리에 비해 훨씬 빠른 접근 시간을 가지며, 높은 대역폭을 얻기 위해 동시에 접근할 수 있는 여러 뱅크(bank)들로 구성되어 있다[10,11]. n개의 뱅크로 이루어진 공유 메모리에서 서로 다른 뱅크에 대한 접근은 동시에 처리되며, 따라서 단일 뱅크의 대역폭 보다 n배 높은 메모리 대역폭을 제공한다[10]. 반면, 같은 뱅크에 두 개 이상의 메모리 요청이 동시에 발생하면, 뱅크 충돌(bank conflict)이

일어났다고 하며, 하드웨어에 의해 순차적으로 처리되어진다[10]. 하나의 뱅크에 동시에 m개의 메모리 요청이 발생한 경우 이를 m-*way* 뱅크 충돌이 발생했다고 하며, 최적의 성능을 얻기 위해서는 뱅크 충돌을 최소화시켜야 한다[10]. 최신 GPU에서 공유 메모리는 총 16개의 뱅크로 이루어지며, 연속적인 32-비트(4-바이트) 워드들은 연속적인 뱅크들에 할당된다[10]. 워프에 의해 요청된 공유 메모리 접근은 두 개의 하프 워프로 나뉘어 독립적으로 처리되며, 그 결과 두 하프 워프에 속한 스레드 간에는 어떠한 뱅크 충돌도 발생하지 않는다[10].

그림 9는 스카이라인 연산에서 사용되는 객체들이 어떻게 공유 메모리에 할당되고, 이에 따라 발생할 수 있는 뱅크 충돌에 대해 나타낸다. 한 객체는 4-바이트 크기의 부동 소수점 형식의 속성들로 구성되므로 워프에 의해 읽혀진 객체들은 그림 9와 같이 각 뱅크에 순차적으로 할당되어지며, 하프-워프의 스레드들은 객체 지배 관계 판별을 위해 객체의 첫 번째 속성부터 순차적으로 공유 메모리에 접근하게 된다. 그림 9의 (a)와 같이 2개의 속성으로 이루어진 객체의 경우 하프-워프의 두 개의 스레드가 동시에 각 뱅크에 접근하게 되므로, 2-way 뱅크 충돌을 발생시킴을 알 수 있다. 반면, 그림 9의 (b)와 같이 한 객체가 3개의 속성으로 이루어진 경우 공유 메모리에 접근하는 하프-워프의 스레드들은 서로 다른 뱅크에 접근하므로 메모리 대역을 매우 효율적으로 활용할 수 있다. 이처럼 스카이라인 처리를 위해 사용되는 객체 집합은 그 속성의 수가 짝수인 경우 뱅크 충돌을 발생시키는 특성을 나타내며, 특히 8의 배수 개의 속성을 갖는 경우 8-way 또는 16-way 뱅크 충돌을



(a) 2-way 뱅크 충돌이 발생하는 경우



(b) 뱅크 충돌이 발생하지 않는 경우

그림 9 공유 메모리 할당 및 뱅크 충돌

발생시켜 메모리 대역을 매우 낭비하게 된다. 따라서 공유 메모리의 메모리 대역을 효율적으로 활용하기 위해 속성의 수가 짝수인 경우 임의의 패딩 값(padding value)을 마지막 속성에 추가하여, 발생할 수 있는 뱅크 충돌을 막고 GPU에서 제공하는 공유 메모리 대역을 최대한 활용할 수 있도록 한다.

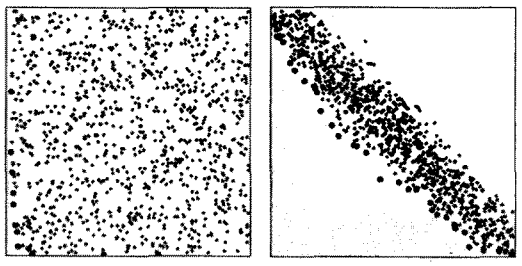
5. 실험 결과

본 장에서는 병렬 중첩-반복 스카이라인 알고리즘이 각 단계별 최적화를 통해 얻을 수 있는 성능 향상을 실험을 통해 나타낸다. 실험에는 엔비디아의 계산 전용 그래픽 카드인 테슬라 C1060이 사용되었다. 테슬라 C1060은 엔비디아 최신 그래픽 칩셋인 GT200 계열로 30개의 멀티프로세서에 각 8개의 스트림 프로세서가 장착되어 총 240개 코어로 동작한다. 또한 삼각함수 등의 계산에 사용되는 2개의 SFU 유닛, 1개의 배정도 정밀도 연산을 처리하는 DP 유닛, 그리고 1만 6천개의 32-비트 레지스터와 16KB 크기의 공유 메모리를 가진다. 레지스터는 개별 스레드들에 할당되어 연산 처리를 위해 사용되며, 공유 메모리는 각 블록에 할당되어 블록 안의 스레드들에 의해 공유되어 사용된다. 표 1은 테슬라 C1060의 제원을 나타낸다.

표 1 테슬라 C1060

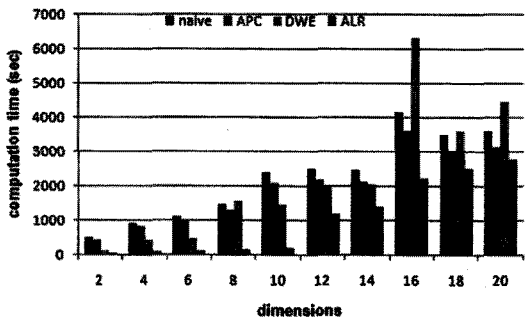
Processor	1 × Tesla T10
Number of cores	240
Core Clock	1.296 GHz
Floating Point	933 Gflops SP
	78 GFlops SP
On-board memory	4.0 GB
Memory bandwidth	102 GB/sec peak
memory I/O	512-bit, 800MHz GDDR3
Form factor	Full ATX
	Dual slot wide
System I/O	PCIe x16 Gen2
Typical power	160 W

병렬 중첩-스카이라인 알고리즘과 단계별 최적화를 통해 얻을 수 있는 성능 측정을 위해 두 가지 형식의 데이터를 사용한다. 그림 10의 (a)는 객체들이 전체적으로 고르게 분포되어 있는 independent 분산 형식을 나타내며, (b)는 객체들이 특정 속성에 편중되어 분포되어 있는 anti-correlated 분산 형식을 나타낸다. 볼드체로 표현된 점들은 스카이라인 집합을 의미한다. 실험에서 사용한 객체 집합의 각 속성들은 실험의 간결성을 위해 4-바이트 크기의 부동 소수점으로 표현되며, 한 객체는 스카이라인 연산에서 고려되는 속성들로만 표현되어 있다.

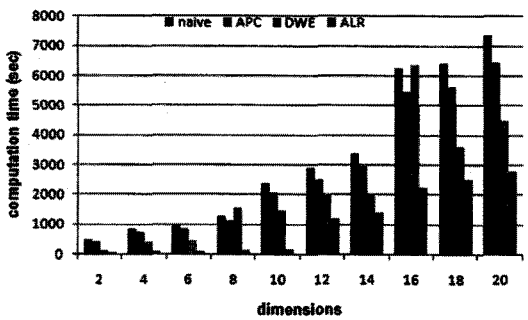


(a) independent 분포 (b) anti-correlated 분포

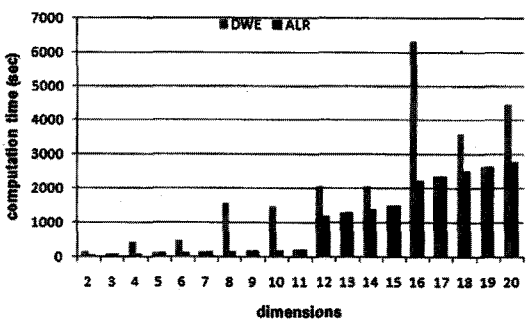
그림 10 데이터 분포



(a) independent 분포



(b) anti-correlated 분포



(c) anti-correlated 분포

그림 11 실험 결과

그림 11은 병렬 중첩-반복 스카이라인 알고리즘과 각 최적화 단계를 통해 얻은 성능 향상을 실험을 통해 나타낸 것이다. 이를 간략하게 설명하면, naive는 기본 중첩-반복 스카이라인 알고리즘, APC(Access Pattern Change)는 전역 메모리 접근 패턴 변경을 통한 메모리 처리율 향상, DWE(Divergent Warp Elimination)는 흐름 제어 명령어 제거를 통한 명령어 처리율 향상, 그리고 ALR(Attribute Layout Reorganization)은 짝수개의 속성을 갖는 객체의 마지막 속성으로 패딩 값을 추가하여 뱅크 충돌을 제거한 최적화 알고리즘을 나타낸다. 그래프의 가로축은 객체의 속성 수, 세로축은 스카이라인 처리를 위해 사용된 수행 시간을 초 단위로 표현했으며, 실험 결과는 전체 객체 집합 수가 1,000,000개(1000k)인 데이터에 대한 결과를 나타낸다.

5.1 전역 메모리 접근 패턴 변경(APC)

전역 메모리의 메모리 대역을 낭비 없이 효과적으로 사용하기 위해 전역 메모리 접근 패턴을 변경한 경우 워프의 스레드들은 객체 단위가 아닌 속성 단위로 접근하기 때문에 최소한의 메모리 트랜잭션을 발생시켜 전역 메모리 대역을 최대로 활용한다. 그림 11의 실험 결과를 통해 *naive*한 처리 방식에 비해 APC의 수행 시간은 (a) independent 분포의 경우 평균 11%, (b) anti-correlated 분포의 경우 평균 12% 향상되었음을 알 수 있다. 이를 통해 적은 수의 메모리 트랜잭션을 발생시켜 효율적으로 전역 메모리 대역을 활용하는 것이 프로그램 성능에 상당한 영향을 줄 수 있음을 알 수 있다. 또한 쿠다 프로그램 구현 시 전역 메모리에 대한 접근 패턴을 얼마나 효과적으로 하는가에 대한 중요성도 알 수 있다. 성능 향상 정도는 전체 데이터 집합의 수와 형식에 상관없이 비슷한 결과를 얻었다.

5.2 흐름 제어 명령어 제거(DWE)

다음 최적화 단계는 분기하는 워프가 발생하는 것을 막기 위해 기본적인 스카이라인 알고리즘을 논리 연산자(*and, or, xor*)를 이용하여 처리함으로서 SIMT로 동작하는 쿠다 특성에 맞게 최적화한 것이다. *naive* 알고리즘은 CPU의 특성을 극대화하기 위해 속성 간의 비교 중 더 이상 진행할 필요가 없을 경우 *break* 명령에 의해 즉시 중단하고 제어의 흐름을 바꾼다. 반면, 변경된 DWE 알고리즘은 분기하는 워프의 발생을 없애기 위해 두 객체의 비교 시 첫 속성부터 마지막 속성까지 모두 비교 연산을 수행하여 처리하는 차이가 있다.

그림 11을 보면 대부분의 경우 상당한 성능 향상을 얻을 수 있지만 속성의 수가 8개, 16개인 경우 *naive*알고리즘보다도 좋지 않은 성능을 나타낸다. 그 이유는 하프 워프의 16개 스레드가 공유 메모리 접근 시 8-way 및 16-way 뱅크 충돌을 발생시켜 공유 메모리에 대한 요청을 대부분 순차적으로 처리하기 때문이다. 이 경우 오히려 CPU에서 사용되는 *naive* 알고리즘을 사용하는

것이 더 낮은 성능을 보이는데, 그 이유는 최소 2개의 속성 비교를 통해 객체 간의 지배 관계가 판가름 날 수 있어 분기하는 워프가 발생하더라도 뱅크 충돌이 발생하는 비효율적인 연산을 줄이는 것이 성능 향상에 도움이 되기 때문이다. 나머지 경우 그림 11의 실험 결과를 통해 APC에 비해 DWE의 수행 시간은 (a) independent 분포의 경우 평균 28%, (b) anti-correlated 분포의 경우 평균 36% 향상되었음을 알 수 있다. 따라서 다소 불필요한 연산을 수행하더라도 SIMT로 동작하는 스레드들이 서로 다른 실행 경로를 가지는 분기하는 워프를 최소화하는 것이 전체 프로그램 성능에 상당한 효과를 줄 수 있음을 알 수 있다.

5.3 공유 메모리 뱅크 충돌 제거(ALR)

마지막 단계는 공유 메모리 상에서 속성의 수에 따라 발생하는 뱅크 충돌을 제거하고 공유 메모리 대역을 최대로 활용할 수 있게 최적화한 것이다. 공유 메모리는 효율적이고 빠른 메모리 접근을 위해 뱅크로 구성되어 있지만 스카이라인의 기본 처리 단위인 객체는 속성의 수가 짝수 개인 경우 뱅크 충돌을 발생시킨다. 특히 속성의 수가 8의 배수인 경우 지배 관계 판별을 위해 접근 되는 객체들의 속성이 같은 뱅크에 밀집되어 최악의 성능을 보인다.

그림 11의 (c)의 실험 결과를 보면 DWE의 경우 짝수 개의 속성을 갖는 데이터가 속성이 하나 더 많은 홀수 개의 속성을 갖는 데이터 보다 좋지 않은 성능을 나타냄을 알 수 있다. 홀수 개의 속성을 갖는 객체 집합의 경우 짝수 개의 속성을 갖는 객체 집합에 비해 더 많은 비교 연산과 메모리 대역을 소모하지만, 뱅크 충돌이 발생하지 않아 상대적으로 더 낮은 성능을 보임을 알 수 있다. 이런 문제를 해결하기 위해 짝수 개의 속성을 갖는 객체의 마지막 속성으로 패딩 값을 삽입하여 발생할 수 있는 뱅크 충돌을 사전에 방지하였다. 그림 11의 실험 결과를 보면 DWE 알고리즘에 비해 ALR 알고리즘의 수행 시간은 (a) independent 분포의 경우 평균 82%, (b) anti-correlated 분포의 경우 평균 83% 향상되었음을 알 수 있다.

6. 결 론

본 논문에서는 엔비디아의 쿠다 병렬 프로그래밍 모델을 이용하여 기본적인 중첩-반복 스카이라인 알고리즘을 병렬화하고, 단계적으로 최적화 시키면서 GPU의 성능 확장성을 알아보았다. 일반적으로, GPU 처리 구조는 영상 처리 및 컴퓨터 비전 알고리즘 같은 대용량 병렬 처리에 적합하지만, 특정 알고리즘의 고유한 특성을 GPU 환경에 맞게 단계적으로 수정해 감에 따라 상당한 성능 향상을 얻을 수 있음을 알았다. 특히 스카이라인 알고리즘처럼 대용량 데이터를 처리하면서 비교 연산이 많이 발생하는 경우 GPU 특성에 맞게 단계적으로 최적화시키면서 상당한 성능 향상을 얻을 수 있었다. 실험을 통해 알 수 있듯이 naive 알고리즘에 비해 ALR 알고리즘은 independent 분포의 경우 평균 78%, anti-correlated 분포의 경우 평균 83%의 수행 시간이 향상되었으며, 이는 속성의 수가 적을수록 더 큰 폭의 성능 향상을 얻을 수 있었다. 그 이유는 속성의 수가 적을수록 불필요한 속성 간의 비교를 수행하지 않는 naive 알고리즘의 이점이 줄어들기 때문이다. 따라서 쿠다 프로그래밍 시 GPU에서 사용할 수 있는 자원은 제약이 있기 때문에 최적의 성능 효율을 얻기 위해서는 알고리즘 특성과 GPU의 하드웨어적 특성을 적절히 조화하여 GPU 자원을 효율적으로 사용해야함을 알 수 있다.

참 고 문 헌

[1] NVIDIA Corporation, http://www.nvidia.com

[2] J. Krueger, R. Westermann, Linear, "Linear algebra operators for GPU implementation of numerical algorithms," In *Proceedings of SIGGRAPH*, pp.908-916, 2003.

[3] J. Bolz, I. Farmer, E. Grinspun, P. Schroeder, "Sparse matrix solvers on the GPU: Conjugate gradients and multigrid," In *Proceedings of SIGGRAPH*, pp.917-924, 2003.

[4] Mark J. Harris, Greg Coombe, Thorsten Scheuermann, and Anselmo Lastra, "Physically-Based Visual Simulation on Graphics Hardware," *Proc. 2002 SIGGRAPH*.

[5] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Kruuger, Aaron E. Lefohn, and Timoty J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," In *Eurographics 2005, State of the Art Reports*, pp.21-51, August 2005.

[6] GPGPU, http://gpgpu.org

[7] Mark D. Hill, Michael R. Marty, "Amdahl's Law in the Multicore Era," *IEEE Computer Society*, 2008.

[8] K. Asanovic et al., "The Landscape of Parallel Computing Research: A View from Berkeley," *report UCB/EECS 2006*, p.183, 2006.

[9] DevNote, "멀티프로세서 프로그래밍 시대의 개막," http://devnote.net/wiki/index.php/Main_Page

[10] NVIDIA CUDATM Programming Guide Version 3.0, NVIDIA Corporation, Santa Clara, CA, USA, 2010.

[11] David Kirk and Wen-mei Hwu, CUDA Textbook, Draft Version, 2009.

[12] R. Rost, OpenGL Shading Language Second Edition, Addison-Wesley, 2006.

[13] J.-M. Frahm, M. Pollefeys, and M. Shah, *Proc. of CVPR Workshop on Visual Computer Vision on*

GPU's, June, 2008.
[14] A. Gopalakrishnan and A. Sekmen, "Vision-based Mobile Robot Learnig an Navigation," ROMAN, *IEEE International Workshop on Robots and Human Interactive Communication*, pp.28-53, 2005.
[15] Stephan Borzsonyi, Donald Kossamann, and Konrad Stocker, "The Skyline Operator," *in ICDE*, pp. 421-430, 2001.
[16] Sungwoo Park, Taekyung Kim, Johghyun Park, Jinha Kim, and Hyeonseung Im, "Parallel Skyline Computation on Multicore Architectures," *in ICDE*, pp.760-771, 2009.
[17] P. Wu, C. Zhang, Y. Feng, B. Y. Zhao, D. Agrawal, and A. E. Abbadi, "Parallelizing skyline queries for scalable distribution," *in EDBT*, pp.112-130, 2006.
[18] A. Cosgaya-Lozano, A. Rau-Chaplin, and N. Zeh, "Parallel computation of skyline queries," *in HPCS*, p.12, 2007.
[19] D. Kossmann, F. Ramsak, and S. Rost, "Shooting stars in the sky: an online algorithm for skyline queries," *in VLDB*, pp.275-286, 2002.
[20] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "Progressive skyline computation in database systems," *ACM Transactions on Database Systems*, vol.30, no.1, pp.41-82, 2005.
[21] Joachim Selke, Christoph Lofi, and Wolf-Tilo Balke, "Highly Scalable Multiprocessing Algorithms for Preference-Based Database Retrieval," *15th International Conference on Database Systems for Advanced Applications (DASFAA)*, Tsukuba, Japan, 04/2010.
[22] S.-R. Cho, H. Han, S.-W. Lee, "Multi-Dimensional Record Scan with SIMD Vector Instructions," *Journal of KIISE : Computing Practices and Letters*, vol.16, no.6, pp.732-736, June. 2010. (in Korean)
[23] J. Chhugani, W. Macy, A. Baransi, A. Nguyen, M. Hagog, S. Kumar, V.W. Lee, Y. K. Chen, and P. Dubey, "Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture," *Proc. of the Very Large Data Base Endowment*, vol.1 issue2, August 2008, pp.1313-1324, 2008.
[24] 민준, 한환수, 이상원, "Multi-core 환경에서 입력 데이터 크기에 따른 skyline 알고리즘 병렬화 고찰", *한국정보과학회 가을 학술발표논문집*, 제 36권 제 2호 pp. 22-23, 2009.

민 준
2003년 가톨릭대학교 컴퓨터공학(학사) 2009년~현재 성균관대학교 정보통신공학부 석사과정. 관심분야는 병렬 처리, GPGPU

한 환 수
1993년 서울대학교 컴퓨터공학(학사). 1995년 서울대학교 컴퓨터공학(석사). 2001년 Univ of Maryland 전산학(박사). 2001년~2002년 Intel 책임연구원. 2003년~2008년 KAIST 전산학과 교수. 2008년~현재 성균관대학교 정보통신공학부 교수 관심분야는 최적화 컴파일러, 고성능 컴퓨팅, 프로그램 분석

이 상 원
1991년 서울대학교 컴퓨터학과(학사). 1994년 서울대학교 컴퓨터학과(석사). 1999년 서울대학교 컴퓨터학과(박사). 1999년~2001년 한국 오라클. 2001년~2002년 이화여대 BK21 계약교수. 2002년~현재 성균관대학교 정보통신공학부 교수. 관심분야는 flash memory DBMS