JDCS
디지털콘텐츠학회논문지
Journal of Digital Contents Society
Vol. 18, No. 4, pp. 739-745, July. 2017

병렬 Shifted Sort 알고리즘의 Warp 단위 CUDA 구현 최적화

박 태 정
덕성여자대학교 디지털미디어학과

Optimization of Warp-wide CUDA Implementation for Parallel Shifted Sort Algorithm

Taejung Park
Department of Digital Media, Duksung Women's University, Samyangro 144gil 33, Dobong-gu Seoul 01369, Korea

[요 약]

본 논문에서는 GPU 병렬 처리 하드웨어 아키텍처 내 최소 물리적 스레드 실행 단위(warp) 내에서 shifted sort 기반 k개 최근접 이웃 검색 기법을 구현하는 방법을 논의하고 일반적으로 동일한 목적으로 널리 사용되는 GPU 기반 kd-tree 및 CPU 기반 ANN 라이브러리와 비교한 결과를 제시한다. 또한 많은 애플리케이션에서 k가 비교적 작은 값이 필요한 경우가 많다는 사실을 고려해서 k가 warp 내부에서 직접 처리 가능한 2, 4, 8, 16개일 때 최적화에 집중한다. 구현 세부에서는 사용한 CUB 공개 라이브러리의 루프 내 메모리 관리 방법, GPU 하드웨어 직접 명령 적용 방법 등의 최적화 방법을 논의한다. 실험 결과, 제안하는 방법은 기존의 GPU 기반 유사 방법에 비해 데이터 지점과 질의 지점의 개수가 각각 2^{23}개 일 때 16배 이상의 빠른 처리 속도를 보였으며 이러한 경향은 처리해야 할 데이터의 크기가 커지면 더욱 더 커지는 것으로 판단된다.

[Abstract]

This paper presents and discusses an implementation of the GPU shifted sorting method to find approximate k nearest neighbors which executes within "warp", the minimum execution unit in GPU parallel architecture. Also, this paper presents the comparison results with other two common nearest neighbor searching methods, GPU-based *k*d-tree and ANN (Approximate Nearest Neighbor) library. The proposed implementation focuses on the cases when k is small, i.e. 2, 4, 8, and 16, which are handled efficiently within warp to consider it is very common for applications to handle small k's. Also, this paper discusses optimization ways to implementation by improving memory management in a loop for the CUB open library and adopting CUDA commands which are supported by GPU hardware. The proposed implementation shows more than 16-fold speed-up against GPU-based other methods in the tests, implying that the improvement would become higher for more larger input data.

색인어 : CUDA, GPGPU, 공간 채움 곡선, 이동 정렬, 대략적 k개 최근접 이웃 검색

Key word : CUDA, GPGPU, Space-filling Curve, Shifted Sort, *k* Approximate Nearest Neighbor Search

http://dx.doi.org/10.9728/dcs.2017.18.4.739

Received 11 July 2017; **Revised** 25 July 2017
Accepted 28 July 2017

***Corresponding Author; Taejung Park**

Tel: +82-02-901-8339
E-mail: tjpark@duksung.ac.kr

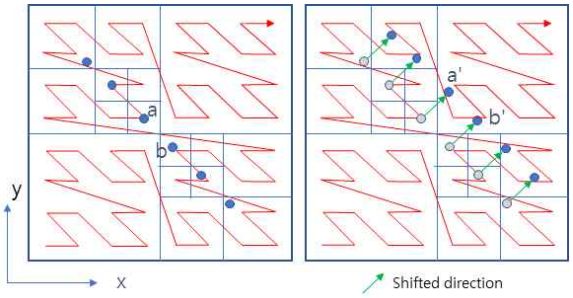


그림 1. kANN shifted sort 알고리즘 개요 [1]
Fig. 1. Overview of kANN shifted sort algorithm [1]

Ⅰ. Shifted sort 기반 대략적 k개 최근접 이웃 탐색 기법

1-1 개요

1) 일반적인 shifted sort 기법

Chan이 발표한 이론([2]의 Lemma 3.3)은 3차원 공간 상에서 데이터 지점과 질의 지점에 대해 공간 채움 곡선(space-filling curve)을 구성하고 5회 평행 이동과 정렬을 수행하면 높은 정밀도로 대략적 k개 최근접 이웃 검색(*k* approximate nearest neighbor search, kANN)을 수행할 수 있음을 증명하였다.

Li et al. [1]이 발표한 shifted sort 알고리즘은 이 이론을 토대로 실제 GPU 병렬 알고리즘을 발표하였다. Shifted sort에 대한 수학적인 증명은 본 논문의 범위를 넘어서는 광범위한 내용이나 [1]에서 제시한 그림(그림 1에서 새롭게 표현)을 통해 개략적인 작동 원리를 이해할 수 있다.

그림 1에서 점들은 데이터 지점 또는 질의 지점을 의미한다. 그리고 격자는 2차원 공간에 대한 쿼드트리(quadtree) 노드들을 나타내고 있다. 왼쪽 아래에서부터 시작해서 오른쪽 위로 향하는 빨간색 선은 공간 채움 곡선들 중 하나인 Morton 코드[3] 곡선을 의미한다. 특히 이 Morton 코드의 진행 방향은 쿼드트리의 너비 우선 탐색(breadth-first traversal)과 동일하다. 이렇게 2차원, 3차원, 혹은 다차원 벡터 공간을 공간 채움 곡선으로 표현하면 다차원 데이터들을 1차원 곡선 위의 정보로 간단하게 표시할 수 있다는 장점이 있다. 그림 1의 왼쪽 사각형 내 점들 중 a와 b는 실제 거리에서는 서로 가깝지만 Morton 코드의 진행 순서(빨간색 선)를 보면 멀리 떨어져 있어서 실제 거리의 정보가 반영되지 않은 것을 볼 수 있다. 이러한 상황은 쿼드트리나 옥트리(octree)에서 서로 다른 부모에 속한 자식 노드 내에 포함된 기하 정보가 실제 거리는 가깝지만 트리 구조 상에서 인접했는지의 여부를 파악하기 힘든 문제와 본질적으로 동일하다. 그러나 그림 1의 오른쪽 사각형 내에서처럼 이 점들을 일정한 거리로 이동(shift)시킨 후 Morton 코드를 순서대로 추적해 보면 이동한 점 a'과 b'이 Morton 코드 진행을 따라 서로 가까워져 있음을 볼 수 있다. Chan은 앞서 언급한 이론([2]의 Lemma 3.3)을 통해 3차원 공간에서는 바운딩 박스 범위 내에서 5회 이동(shift)과 정렬을 반복해서 가장 가까운 점들을 계산하면 매우 높은 정밀도로 k개의 최근접 이웃을 탐색할 수 있음을 증명한 것이다.

Li et al. [1]은 이러한 이론을 바탕으로 GPU병렬 shifted sort 알고리즘을 발표하였으나 대략적인 유사 코드(pseudo code)만 제시되었고 구체적인 구현에 대한 내용은 논의되지 않았다. 또한 실제 애플리케이션에서 CUDA 구조의 최소 물리적 병렬 처리 단위인 warp에서 한 번에 병렬 처리가 가능할 정도로 k가 작은 경우가 빈번하게 발생하나 이러한 문제에 대한 최적화를 논의하지 않았다. 따라서 본 논문에서는 [1]이 제안한 shifted sort 알고리즘을 NVIDIA CUDA 환경에서 구현하는 실제와 warp 단위 연산의 최적화 방안을 논의하고 그 결과를 제시하고자 한다.

2) Shifted sort 기반 최근접 이웃 검색 기법의 장점

병렬 shifted sort 알고리즘은 기존의 최근접 이웃 검색 기법으로 널리 사용되는 GPU 기반 *k*d-tree에 비해 상당히 우수한 성능을 얻을 수 있다. 본 논문에서 논의하는 실험 결과 GPU 기반 *k*d-tree에 비해서 최대 약 16 배 정도 빠른 처리 속도를 보였으며 입력 데이터의 크기가 더 커지면 이 차이는 더 커질 것으로 예상된다.

이러한 실행 시간의 단축 원인에는 몇 가지가 있으나, 가장 중요한 요인은 kd-tree를 포함해서 일반적인 트리 탐색 시 한 하드웨어 실행 단위(warp) 내부에서 스레드들 마다 다른 분기문(if 문) 브랜치를 실행해야 하는 warp divergence[4]가 필연적으로 일어나서 성능을 저하시키는 것에 비해서 shifted sort 방식은 탐색 시 분기문이 거의 사용되지 않으며 이미 최적화가 상당한 수준에 오른 병렬 정렬(sort) 기술을 그대로 이용할 수 있다는 특성이라고 할 수 있다.

Ⅱ. Warp 단위 병렬 Shifted Sort 알고리즘

2-1 Warp 단위 병렬 shifted sort 알고리즘의 필요성

I 장에서는 Li et al. [1]이 발표한 k가 충분히 큰 일반적인 kANN 알고리즘에 대해 논의하였다. 이 저자들은 포톤 스캐터링(photon scattering)과 같은 전역 조명(global illumination)에 적용하기 위해서 k 값이 warp 크기인 32를 초과할 때에 주로 집중해서 논의했다. [1]에서 32 이하인 경우도 계산되었지만 (예. [1]의 Figure 6) 이 경우의 최적화 – 특히 하드웨어 특성을 활용한 최적화 – 에 대한 언급은 논의 되지 않았다.

전역 조명과 같이 k가 큰 경우 중요한 적용 분야도 존재하지만 3차원 매시(mesh) 주변의 거리장 계산 등과 같은 여러 3차원 기하 정보 탐색에서는 k가 클 경우, 오히려 여러 후보들 중에서 다시 가장 조건에 적합(예를 들어 최근접)한 요소를 다시 탐색해야하기 때문에 k가 4~8 정도일 때 연산 효율이 가장 높다. 따라서 일반적인 3차원 기하 정보 탐색과 같은 문제에서는 k가 4~8 정도일 때 최적화된 알고리즘이 필요하다.

이러한 측면에서 본 논문에서는 k가 32 미만의 값을 가지는

대략적 최근접 탐색(approximate nearest neighbor search) 알고리즘의 최적화 방안을 논의한다.

2-2 Warp 단위 병렬 shifted sort 구현

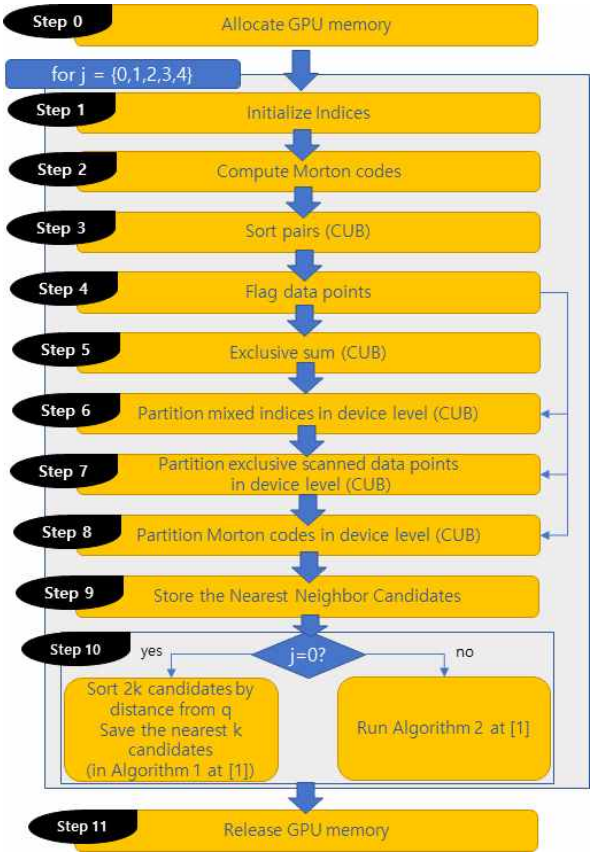


그림 2. Warp 기반 shifted sort의 세부 단계
Fig. 2. Detailed steps in warp-based shifted sort

그림 2에서는 본 논문에서 제안하는 warp 기반 shifted sort 알고리즘의 흐름을 제시한다. 전체적으로는 [1]에서 제안한 방식에 기초하고 있으나 [1]에서는 대략적인 유사 코드만 제시되었고 warp 내부에서의 최적화 방안이 논의되지 않았으며 특히 논문이 발표되던 시기 혹은 그 이후에 등장한 GPGPU의 하드웨어 및 소프트웨어의 최적화 기술이 반영되지 않은 측면이 있다. 따라서 본 논문에서 제안하는 구현 방법은 CUDA 코드 구현 및 CUB나 Thrust 같은 라이브러리의 사용과 warp 내부에서의 영향과 작동 방식에 대해 좀 더 세부적으로 논의한다.

그림 2의 각 단계는 모두 NVIDIA사의 CUDA[5] 로 구현되었으며 (CUB)로 표시된 단계는 NVIDIA Research 소속인 Duane Merrill이 제작해서 공개하는 CUB 라이브러리[6]을 사용해서 구현되었으며 나머지는 일반적인 CUDA 커널[5]로 작성되었다. Duane Merrill의 CUB 라이브러리는 공개 소프트웨어 형식으로 업데이트되고 있으며 warp 범위에서부터 block, device 범위에 이르는 병렬 연산 처리 단위를 대부분 지원하며 최신 아키텍처를 포함하는 여러 CUDA 하드웨어 아키텍처에 최적화된 다양한 기본 병렬 알고리즘을 제공한다. 성능적인 측면에서 GPU 자원 관리의 용이성과 처리 속도 최적화 등의 장점 뿐만 아니라 개발 과정에서 C++ 제너릭 프로그래밍, Reflective 클래스 인터페이스 등 CUDA에서 접근하기가 쉽지 않은 고급 객체지향기법으로 소스 코드의 개발과 관리를 지원하는 장점이 있다.

최근부터는 CUDA 개발 도구 (CUDA Toolkit) [7] 공식 배포판에 포함된 Thrust 라이브러리를 내부적으로 구동하는 기본 코드로 CUB가 적용된다. 개발자가 GPU 하드웨어 특성을 거의 신경 쓰지 않고 사용할 수 있는 Thrust 라이브러리에 비해 CUB 라이브러리는 세세한 부분을 지정해야 한다는 단점이 있으나, Thrust 라이브러리는 디버깅 작업 시 대부분 내부에서 발생하는 상황을 파악하기 힘든 반면 CUB 라이브러리는 공개 라이브러리이기 때문에 내부 코드 단위로 디버깅이 가능하다는 장점이 있다. 본 논문에서는 세부적인 성능 튜닝과 디버깅을 위해서 Thrust 대신 CUB를 사용하였다.

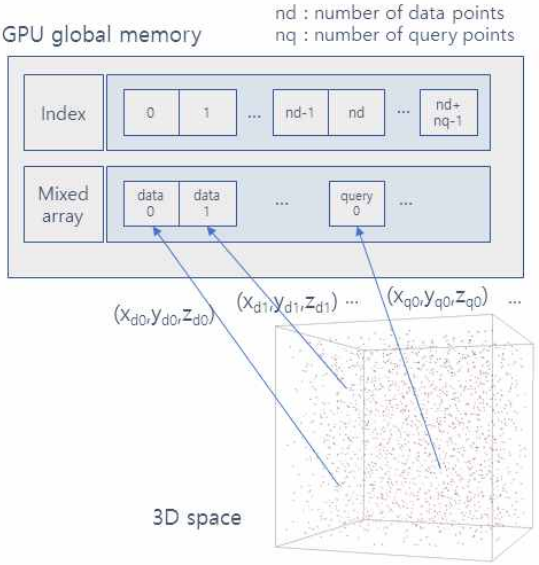


그림 3. 데이터 및 질의 지점 혼성 배열 구조
Fig. 3. Structure of Mixed array combining data points and query points

2-3 각 단계의 설명

1) GPU 메모리 할당

그림 2에서 Step 0으로 표시된 단계에서는 데이터 지점의 개수와 질의 지점의 개수를 파악해서 GPU 내부에 연산을 위한 메모리를 확보한다. 표 1의 가장 오른쪽 열에서 각 데이터 지점 및 질의 지점 개수에 따른 GPU 메모리 사용량을 제시했다. 이 결과에서 확인할 수 있는 것처럼, 제안하는 구현에서 GPU 메모리 사용량은 각 데이터 지점 및 질의 지점 개수에 선형적으로 비례한다.

2) 색인 초기화 및 데이터/질의 정보의 단일 배열 저장

Step 1에서는 GPU 메모리 내 색인 배열을 할당하고 병렬로 색인값을 0에서부터 (데이터 지점 개수 + 질의 지점 개수) -1 까

지 할당한다. 제안하는 구현에서는 병렬 처리 효율의 극대화를 위해서 데이터 지점과 질의 지점을 GPU 내 동일한 배열 하나에 저장한 후 처리하며 필요에 따라 생성한 플래그 배열(Step 4)을 이용해서 데이터와 질의를 병렬 연산으로 분리한다.

초기화 시에는 배열 색인 0에서부터 (데이터 지점 크기) - 1까지 3차원 데이터 지점 값이 들어가고 그 이후 공간에는 3차원 질의 지점 값이 저장된다(그림 3).

3) 64비트 Morton 코드 생성

Step 2에서는 Step 1에서 생성한 데이터/질의 혼성 배열(mixed array, 그림 3) 내 3차원 공간상의 점 (x,y,z)에 대해 [3]에서 제안한 64비트 버전 Morton 코드를 병렬로 생성한다. 이렇게 생성한 Morton 코드는 별도의 GPU 내 배열에 저장한다.

4) 혼성 배열 정렬(CUB)

직전 단계에서 계산한 Morton 코드 배열에는 배열 색인 기준으로 앞쪽(색인 0)부터 데이터 지점의 Morton 코드가 저장되고 그 다음 연속으로 질의 지점의 Morton 코드가 저장된다. Step 3에서는 이렇게 구성된 Morton 배열을 CUB의 device 범위 radix 정렬(cub::DeviceRadixSort::SortPairs)을 이용해서 정렬한다. [3]에서 논의한 대로, Morton 코드를 정렬한 결과는 3차원 공간 내 데이터 지점과 질의 지점을 대상으로 옥트리(octree) 분할을 수행한 후 너비 우선 탐색(breadth first search) 순서대로 재정리한 결과에 해당된다.

CUB 라이브러리 내 GPU 병렬 처리 함수는 GPU 메모리 관리를 위해 독특한 방법을 적용한다. 즉, 1) 먼저 사용하고자 하는 CUB 함수를 호출해서 실제 필요한 GPU 메모리를 계산한 후 2) 그 크기만큼 메모리를 실제로 할당하고 그 다음 3) 처음과 동일한 함수를 다시 호출한다.

예를 들어 cub::DeviceRadixSort::SortPairs 함수의 경우 다음과 같은 표준적인 코드 구성을 적용한다(첫번째 줄과 세 번째 줄의 코드는 완전히 동일함).

```
SortPairs(storage, storage_bytes, morton, ...);
cudaMalloc(&storage, storage_bytes);
SortPairs(storage, storage_bytes, morton, ...);
```

그림 2에서 Step 1 ~ Step 10 사이 과정은 [2]에서 증명한 대로 Morton 코드와 같은 space filling curve를 공간 내에서 조금씩 위치를 이동시키면서 정렬을 5회 반복한다. 이러한 반복 구조로 인해서 모든 CUB 함수들을 위에서 제시한것처럼 일반적인 방식으로 사용하면, 실험 결과에서 GPU 메모리의 해제 없이 루프 안에서 계속 메모리가 불필요하게 낭비되는 상황이 발생하였다. 이 구현 문제에서는 루프 내부에서 질의 및 데이터 지점의 크기가 고정되기 때문에 루프 내부에서 한 번만 실행되도록 하였다. 즉,

```
if(j==0){
 SortPairs(storage, storage_bytes, morton, ...);
 cudaMalloc(&storage, storage_bytes);
}
SortPairs(storage, storage_bytes, morton, ...);
```

루프 내 모든 CUB 함수들은 if문을 적용함으로써 루프 진행과 함께 GPU 메모리 소비가 불필요하게 증가하는 상황을 방지할 수 있었다.

5) 데이터/질의 지점 구분을 위한 플래그 정보 병렬 생성

직전 단계에서는 데이터/질의 지점의 Morton 코드를 한 배열에서 정렬하기 때문에 정렬한 Morton 코드에서는 데이터/질의 지점을 구분할 수단이 필요하다. 제안하는 구현에서도 [1]에서 적용한 방식을 사용하였다. 즉, 3차원 데이터를 x, y, z 각각 21비트 씩 할당해서 64비트 Morton 코드를 구성하면 21x3=63비트가 소요되고 남는 1비트를 이용해서 데이터/질의 지점을 구분한다. 그러나 이렇게 마지막 비트를 bitwise 연산으로 확인하는 절차가 번거로우며, 상대적으로 큰 64비트 크기의 Morton 코드 배열을 매번 GPU로 전달해야 하기 때문에 GPGPU 연산 속도를 결정하는 GPU(device)/CPU(host) 사이의 대역폭에 나쁜 영향을 줄 수 있다. 따라서 제안하는 구현에서는 Step 4에서 별도의 플래그 배열을 병렬로 계산한 후에 그 이후의 필요한 병렬 처리 단계(그림 2에서 오른쪽 측면 화살표를 통해 필요한 단계로 연결)에서 이 플래그 배열을 사용하는 방법을 적용하였다. 이 단계의 결과로 데이터 지점들은 1, 질의 지점들은 0으로 설정된 플래그 배열을 얻는다.

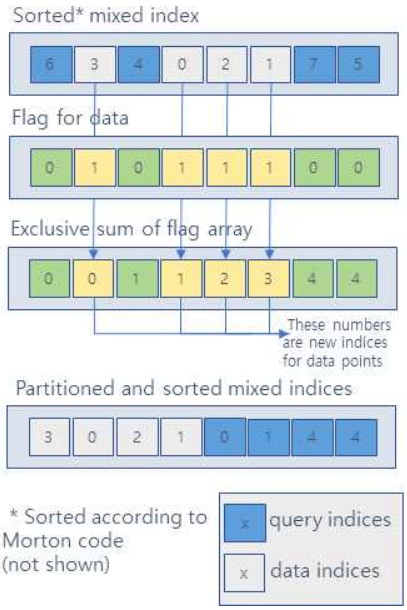


그림 4. 익스클루시브 스캔으로 새로운 배열에서의 위치 계산
Fig. 4. Calculation of new indices using exclusive scan

6) 정렬된 데이터 지점들만 병렬 처리로 모으기 위한 연산

Step 3에서 정렬 연산(radix sort)의 결과로 데이터 지점과 질의 지점의 Morton 코드가 섞여 배열 하나에서 Morton 코드가 낮은 값(octree에서 root node에 가까운 레벨)에서 높은 값(octree에서 leaf node에 가까운 레벨)으로 정렬된다.

Step 5-8은 이렇게 정렬된 배열에서 데이터 지점끼리의 상대적인 선후 관계와 질의 지점에서의 상대적인 선후 관계를 유지한 채 배열 하나에 0에 가까운 배열에는 연속으로 데이터 지점

들을 저장하고 뒤쪽에는 연속으로 질의 지점들을 저장하기 위한 과정이다. 이 과정에서 병렬 익스클루시브 스캔 연산[8]을 이용해서 새로 배치할 배열 색인을 결정하는 작업을 수행한다. 그림 4에서 이 과정을 개략적으로 설명한다. 이전 단계에서 수행한 것처럼 Morton 코드에 대해 정렬을 수행한 후 혼성 색인 배열(그림 4에서 첫 번째 배열(Sorted mixed index))을 얻는다. 이 배열을 이용해서 Step 4를 통해 두 번째 배열(Flag for data)을 얻는다. 이 Flag 배열에 병렬 익스클루시브 합계를 수행하면 플래그를 1로 설정했던 데이터 지점에 해당하는 결과값이 새로 배치할 배열에서의 색인(위치)이 된다.

예를 들어 그림 4에서 데이터 지점 색인인 3은 플래그 배열의 익스클루시브 합계 연산의 결과가 0이 되는데(그림 4의 화살표 참고) 이 값(0)은 최종 배열(Partitioned and sorted mixed indices)에서 데이터 지점 색인 3이 들어갈 위치(색인)가 된다. 마찬가지로 Sorted mixed indcx에서 회색으로 표시된 데이터 색인 0은 익스클루시브 스캔 결과 값이 1이 되고 따라서 최종 배열에서 데이터 색인 0이 배열 색인 1인 위치로 복사된다.

익스클루시브 스캔(합계)을 이용해서 새로운 색인을 계산하는 이 방법은 for 루프를 사용하는 순차적인 연산과 다르게 여러 개의 스레드를 이용해서 병렬적으로 계산해서 연산 속도를 줄일 수 있는 장점이 있다.

7) 각 질의 지점 기준 앞뒤로 가까운 데이터 지점 검색, 저장

Step 9에서는 직전 단계까지 계산한 정보를 이용해서 각 질의 지점에 대해 질의 지점이 정렬된 혼성 색인 내(예를 들어 그림 4의 Sorted mixed index)에서 자신의 위치를 확인하고 이 정보를 이용해서 그림 4의 Partitioned and sorted mixed indices에서 연속된 데이터 지점들에서 해당 질의 지점의 위치(그림 4에서 질의 지점 색인 4의 경우 Partitioned and sorted mixed indices의 색인 0(값 3)과 색인 1(값 0) 사이)를 찾아서 앞뒤로 각각 k개의 인접 데이터 색인 정보를 찾아서 저장한다. 따라서 Step 9에서는 총 2k개의 최근접 후보 색인들이 저장된다.

8) 가까운 k개 지점 저장 및 관리

Step 10에서는 원래 논문[1]에서 제시한 흐름을 그대로 반영하고 있다. j=0일 때에는 2k개의 데이터 지점 후보들을 질의 지점과의 거리에 따라 정렬을 해서 저장한다. 본 논문에서 구현한 방식에서는 [9]의 radix sort를 이용하였으며 특히 [9]에서 논의되지 않은 GPU 하드웨어 지원 함수인 __shfl 함수[10]를 사용하여 warp 단위에서의 성능을 더욱 향상시켰다. j가 0이 아닌 나머지 경우의 알고리즘은 [1]의 Algorithm 2를 구현하였으며 특히 warp 내부에서 연산 효율성을 증가시키기 위해서 GPU 공유 메모리(shared memory)의 활용을 최적화하는 방향으로 구현하였다.

Ⅲ. 결과 분석 및 논의

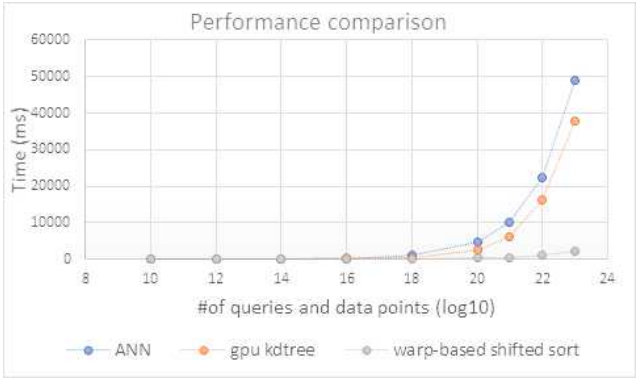
3-1 다른 최근접 알고리즘과의 비교.

표 1에서는 널리 사용되는 다른 최근접 알고리즘인 ANN(Approximate Nearest Neighbor) [11], GPU 기반 kd-tree [12] 와 본 논문에서 제안하는 warp 기반 shifted sort의 결과 비교를 제시한다. 이 실험에서는 단순하게 정규화된 3차원 공간 ($[0.0, 1.0]^3$) 에서 데이터 지점의 개수와 질의 지점의 개수를 동일하게 설정하고 각 점의 개수를 2^{10}개(총합 $2x2^{10}$개)부터 2^{23}개(총합 $2x2^{23}$개)까지의 연산 시간을 측정하였다. NVIDIA 사의 GTX Titan X (메모리 12GB)를 장착한 Intel i7 CPU(3.60GHz), RAM 16GB시스템에서 Windows 7(64bit), CUDA 8.0을 적용한 시스템에서 구현, 테스트를 수행하였다. 또한 실행 코드는 Visual Studio 2012 버전에서 CUDA 코드와 C++ 코드 모두 64비트 release 모드로 컴파일을 수행하였다.

표 1. ANN, GPU 기반 *k*d-tree와 warp 기반 shifted sort 실행 결과 비교 (blocksize = 512)

Table. 1. Performance comparison between ANN, GPU-based kd-tree, and warp-based shifted sort (blocksize = 512)

data size	query size	ANN (ms)	GPU *k*d-tree (ms)	Warp-based shifted sort (k=4, ms)	GPU mem for shifted sort (MB)
2^{10}	2^{10}	1	115	**2**	**0.3**
2^{12}	2^{12}	5	118	**6**	**1.5**
2^{14}	2^{14}	25	145	**8**	**5**
2^{16}	2^{16}	171	213	**21**	**22**
2^{18}	2^{18}	952	552	**71**	**89**
2^{20}	2^{20}	4627	2661	**270**	**356**
2^{21}	2^{21}	10008	5962	**522**	**712**
2^{22}	2^{22}	22516	16053	**1059**	**1424**
2^{23}	2^{23}	48878	38003	**2329**	**2848**



이 실험에서 ANN과 GPU 기반 *k*d-tree는 질의 지점 하나에 대해 가장 가까운 데이터 지점 1개만을 찾은 것에 비해서 제안하는 warp 기반 shifted sort는 가장 가까운 순서대로 4개(k=4)를 찾은 결과를 제시하였다. 표 1의 결과를 보면, ANN 및 GPU 기반 *k*d-tree의 경우 데이터 지점과 질의 지점의 개수가 증가할수록 실행 시간 차이가 크게 증가함을 볼 수 있으나(표 1 내 그래프에서 x축은 log 스케일 적용) 제안하는 warp 기반 shifted

sort는 실행 시간이 매우 완만하게 증가하며 절대적인 시간 역시 크게 증가하지 않음을 볼 수 있다. 특히 데이터 지점과 질의 지점의 개수가 2^{23}개인 경우 실행 시간이 각각 약 17배(ANN 대비), 13배(GPU 기반 *k*d-tree 대비) 차이가 난다는 사실을 확인할 수 있다.

더군다나 3-2절에서 논의한 것처럼 k값이 줄어들면 warp 기반 shifted sort의 실행 시간도 줄어든다는 사실을 감안한다면 제안하는 warp 기반 shifted sort는 기존 방식보다 우수한 성능을 제공한다고 할 수 있다.

표 1의 가장 오른쪽 열에는 warp 기반 shifted sort가 사용한 GPU 메모리 크기를 정리하였다. 이 결과를 보면 제안하는 구현이 사용하는 메모리의 크기가 정확하게 데이터 지점 및 질의 지점의 개수와 비례함을 볼 수 있다. 즉, 데이터 및 질의 지점이 각각 2^{20}개에서부터 2^{23}개까지 데이터 및 질의 지점이 2배씩 증가할 때 사용된 GPU 메모리의 크기도 정확하게 2배씩 증가하였다.

3-2 k값에 따른 성능

표 2. *k*값에 따른 warp 기반 shifted sort 실행 시간 (blocksize = 512, 데이터 지점 개수와 질의 지점 개수는 각각 2^{22})

Table. 2. Execution time of warp-based shifted sort according to various *k* values (blocksize = 512, # of data points and query points are 2^{22} respectively)

k	2	4	8	16
Time (ms)	682	1051	1838	3842

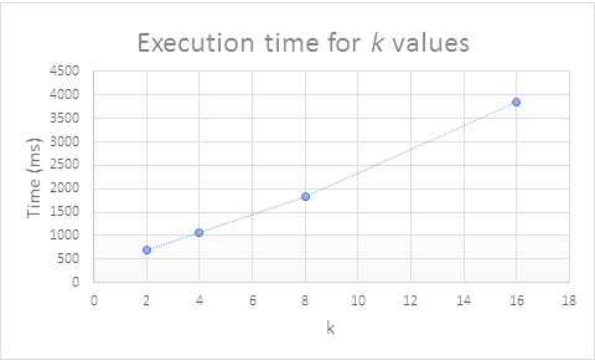


표 2는 데이터 지점의 개수와 질의 지점의 개수가 각각 2^{22}개일 때 warp 기반 shifted sort에서 한 질의 지점에서 가장 가까운 순서대로 k개의 데이터 지점을 찾을 때, k의 변화에 따른 실행 시간을 제시한 것이다. CUDA warp의 크기가 32이고 shifted sort 알고리즘의 특성 상 가장 가까운 k개의 데이터를 찾기 위해서는 2k개를 warp 하나에 넣고 병렬로 검토, 계산해야하기 때문에 적용 가능한 k 값은 $k \in \{2, 4, 8, 16\}$이다.

이 표를 그래프로 나타낸 아래 그림을 보면 $2 \le k \le 16$ 범위에서 k 값에 따라 실행 시간이 거의 선형적으로 변한다는 사실을 확인할 수 있다.

Ⅳ. 결 론

Warp 단위에 최적화된 k 범위(2, 4, 8, 16)의 shifted sort 병렬 연산 기법을 CUDA에서 구현하는 방안을 논의하고 제시하였다. 본 논문에서는 원래 논문에서 논의되지 않은 구현 세부 정보의 실제를 논의할 뿐만 아니라 원래 논문의 발표 이후 등장한 몇 가지 하드웨어/소프트웨어 측면에서의 개선 기능도 적용하여 더욱 성능을 개선하였다.

실험 결과, 제안하는 병렬 shifted sort 구현은 기존의 최근접 이웃 검색 기법으로 널리 사용되는 ANN과 GPU 기반 *k*d-tree에 비해 빠른 우수한 성능을 나타내었다. 실험 결과 GPU 기반 *k*d-tree에 비해서 최대 약 16 배 정도 빠른 처리 속도를 보였으며 입력 데이터의 크기가 더 커지면 이 차이는 더 커지는 경향성을 확인하였다.

현재 이러한 실행 시간의 단축 원인들 중 가장 중요한 요인은 kd-tree를 포함해서 일반적인 트리 탐색 시 한 하드웨어 실행 단위(warp) 내부에서 스레드들 마다 다른 분기문(if 문) 브랜치를 실행해야 하는 warp divergence가 shifted sort에서는 많이 발생하지 않는다는 사실 때문으로 생각하고 있으며 향후 추가 연구를 통해 정량적인 분석을 수행하여 더욱 우수한 병렬 처리 알고리즘을 개발할 계획이다.

감사의 글

본 연구는 산업통상자원부(MOTIE)와 한국에너지기술평가원(KETEP)의 지원을 받아 수행한 연구 과제입니다(과제번호 : 20161210200610).

이 논문은 2016년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임 (NRF-2016R1D1A1B03933660).

참고문헌

[1] S. Li, L. Simons, J. B. Pakaravoor, F. Abbasinejad, J. D. Owens, and N. Amenta, “kANN on the GPU with shifted sorting,” In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics (EGGH-HPG'12)*, Switzerland, pp. 39-47, 2012.

[2] T. M. Chan, “Approximate nearest neighbor queries revisited,” In *Proceedings of the Thirteenth Annual Symposium on Computational Geometry (SCG ’97)*, New York, pp. 352–358, 1997.

[3] T. Park, “Analysis of Morton Code Conversion for 32 Bit IEEE 754 Floating Point Variables,” *The Journal of Digital Contents Society*, Vol. 17, No. 3, pp. 165-172, June 2016.

[4] J. Cheng, M. Grossman, and T. McKercher, Professional CUDA C Programming, 1st ed. Wrox, pp. 84-87, 2014.

[5] CUDA C Programming guide. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz4jURrxaId
[6] CUB official site. Available: https://nvlabs.github.io/cub/#sec1
[7] CUDA Toolkit documentation. Available: http://docs.nvidia.com/cuda/
[8] T. Park, “Correct Implementation of Sub-warp Parallel Prefix Operations based on GPU Hardware Architecture,” *The Journal of Digital Contents Society*, Vol. 18, No. 3, pp. 613-619, June 2017.
[9] Mark Harris, GPU Gems 3, ch. 39. “Parallel Prefix Sum (Scan) with CUDA”. Available: https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch39.html
[10] Shuffle: Tips and Tricks, GPU Technology Conference material. Available: http://on-demand.gputechconf.com/gtc/2013/presentations/S3174-Kepler-Shuffle-Tips-Tricks.pdf
[11] ANN: A Library for Approximate Nearest Neighbor Searching website. Available: https://www.cs.umd.edu/~mount/ANN/
[12] T. Park, “Implementation and Analysis of Parallel kd-Tree Based on Binary Radix Tree with OptiX Realtime Raytracing Framework for Collision Detection and Realtime Raytracing”, Korean Society for Computer Game, vol. 27, No. 3, pp. 53-60, September 2014.

박태정(Taejung Park)

1997년 : 서울대 전기공학부 (공학사)
1999년 : 서울대 전기공학부 대학원 (공학 석사, 반도체 물리 전공)
2006년 : 서울대 전기컴퓨터공학부 대학원 (공학박사, 컴퓨터 그래픽스 전공)

2006년～2013년: 고려대학교 연구교수
2013년～현재 : 덕성여자대학교 디지털미디어학과 조교수
※관심분야 : 컴퓨터그래픽스, 병렬처리, 게임 물리, 수치해석, 3차원 모델링