공학석사 학위논문

배열 기반 데이터베이스를 이용한 대기 스캐닝 라이다 분석 최적화

2021 년 08 월

서울대학교 대학원

컴퓨터공학부

김 주 훈

배열 기반 데이터베이스를 이용한 대기 스캐닝 라이다 분석 최적화

지도 교수 문 봉 기

이 논문을 공학석사 학위논문으로 제출함
2021 년 04 월

서울대학교 대학원
컴퓨터공학부
김 주 훈

김주훈의 공학석사 학위논문을 인준함
2021 년 06 월

위 원 장 _____ 김 형 주 _____

부위원장 _____ 문 봉 기 _____

위 원 _____ 강 유 _____

초 록

대기 스캐닝 라이다 분석은 한 지점에 설치된 라이다를 통해 관측된 특정 반경 안의 결과를 분석하여 해당 지역의 조밀한 대기 농도 결과를 얻어내는 과정이다. 연구자들은 기존에 RDBMS와 GIS를 사용하거나, MATLAB이나 Python 등을 통해 해당 분석 과정을 구현하여 사용하고 있다. 한 대의 라이다 장비에서 관측되는 크기가 작은 데이터에 대해서는 해당 방법들로 가능하지만, 여러 대의 장비에서 관측된 오랜 기간의 데이터에 대해서는 기존 방법들만으로는 분석 성능의 한계가 존재한다. 연구자들이 파라미터를 변경해가며 분석 결과를 확인하는 작업은 빈번하게 발생하기 때문에, 데이터의 특성과 분석 과정에 알맞은 데이터베이스를 사용함으로써 쉽고 빠르게 큰 데이터를 관리하고 분석할 수 있어야 한다.

본 연구에서는 해당 문제를 해결하기 위하여 배열 데이터베이스를 적용하였다. 대기 스캐닝 라이다 데이터는 다차원적 특성을 가지고 있고, 분석과정에서 인접한 cell 사이의 locality를 활용할 수 있기 때문에 배열 기반 데이터베이스의 사용이 적절하다. 또한, SciDB의 shared nothing 구조를 활용하여 높은 scalability를 달성할 수 있다. 대기 스캐닝 라이다 분석과정을 SciDB안의 User-Defined Operator (UDO)를 통해 구현을 하였고, 두 가지 방향을 가지고 성능을 최적화하였다. 첫 번째 방법은 SciDB의 parallel chunk processing을 활용하였다. Operator 수행 시 여러 Instance들 안에 있는 chunk들이 병렬적으로 수행된다. 두 번째 방법은 분석 알고리즘 상에서 반복되는 연산들이 많은 Operator에 대해서 GPU를 통해 병렬화를 시도하였다.

실제 관측된 데이터들을 토대로 실험을 진행하였으며, 기존 방식들과 비교를 하였을 때 배열 기반 데이터베이스의 사용이 분석 성능 면에서 더 효율적임을 보였다. 해당 방법을 통해 연구자들은 짧은 시간 안에 큰 데이터에 대한 분석 결과를 얻을 수 있음을 확인하였다.

주요어 : SciDB, UDO, Parallel Chunk Processing, GPU
학 번 : 2019-26528

목 차

표 목차

그림 목차

제 1 장 서 론

제 1 절 연구의 배경

LiDAR (Light Detection And Ranging)는 레이저를 통해 특정 물체에 반사되어 돌아오는 시간 등을 측정하여 거리를 결정하는 기술이다. 해당 기술을 활용한 다양한 응용이 존재하는데, 자율 주행 자동차에서의 물체 인식, 지형 또는 물체의 표면 탐지, 대기 성분 분석 등이 존재한다. 대기 스캐닝 라이다는 대기 성분을 분석하기 위한 LiDAR 응용 중에 하나이다. 스캐닝 방법을 적용하여 넓은 지역의 대기를 관측하는데 사용된다. 바다에서의 바람 움직임 측정[1]과 도시에서의 대기 오염 관측[2]은 대기 스캐닝 라이다를 이용한 대표적인 예시에 해당한다.

라이다에서 관측된 데이터를 분석하여 대기 분석 결과를 얻는 과정은 크게 세 가지로 구성된다. 첫 번째 과정은 전처리 과정으로, 대기 농도 변환 식의 적용이 가능하도록 Raw data를 수정해주는 작업이다. 예를 들어, Background 노이즈 제거, 거리 보정, Moving Average 적용 등이 해당된다. 두 번째 과정은 실제 농도 변환 식을 적용하는 과정이다. 이는 기존에 과학자들이 연구해 놓은 식들을 토대로 농도 결과를 얻어내는 과정이다. 예를 들어, Klett 1981 method[4]를 활용해서 라이다 신호 데이터를 대기 농도로 변환하는 시도가 있었고, 관련 연구[11]가 계속해서 진행되고 있다. 마지막으로 후 처리 과정이 필요한데, 이는 분석 결과를 시각화하기 위함이다. 위 경도 축에 맞게 coordinate를 변환하는 작업과 결과를 조밀하게 표현하기 위해 interpolation하는 작업이 이에 해당된다.

연구자들은 이러한 분석을 하기 위해서 Relational Database Management System (RDBMS)와 Geographic Information System (GIS)를 사용[3]하거나 MATLAB 또는 Python을 사용하여 분석 과정을 구현하여 사용하고 있다. 데이터의 크기가 작을 때는 이러한 방법들로 분석 결과를 확인하여도 성능에 큰 무리가 없지만, 데이터의 크기가 커지게 되면 기존의 방법들로는 분석 결과를 확인하기까지 많은 시간이 소요되게 된다. LiDAR 네트워크를 통해 더 넓은 관측 범위를 cover하려는 시도[5]는 빅 데이터의 생성을 암시한다. 360도 범위의 5km 반경 관측을 가정했을 때, 남한의 전체 면적을 cover하기 위해서는 약 1200대 정도의 장비가 필요하다. 라이다 장비의 수가 증가하며, 오랜 기간 동안 관측 데이터가 쌓이게 되면, 이들을 한 번에 분석할 때 성능에 bottleneck이 발생한다. 또한, 연구자들이

파라미터를 변경해 가며 분석 결과를 확인하는 작업들은 빈번하게 발생하기 때문에 성능 개선은 필수적이다.

본 연구에서는 이러한 문제를 해결하기 위해 배열 기반 데이터베이스를 사용하였다. 데이터를 분석할 때 적절한 데이터 모델을 사용하는 것은 매우 중요하다. 대기 스캐닝 라이다 데이터는 다음의 두 가지 이유로 배열 데이터베이스의 사용이 적절하다. 첫 번째 이유는 대기 스캐닝 라이다 데이터가 다차원적 특성을 지니기 때문이다. Raw data의 경우 H/W ID, 시간, 각도, 거리 총 4개의 차원들로 이루어져 있으며, 최종 분석 결과 역시 H/W ID, 시간, 위도, 경도 총 4개의 차원들로 구성된다. 기존에 많이 사용되던 RDBMS는 이러한 다차원 데이터를 처리하기에는 부적합하다고 알려져 있다[12]. 두 번째 이유는 분석 과정에 인접한 cell들 사이의 locality가 활용되는 경우가 많기 때문이다. 예를 들어, interpolation 같은 연산의 경우 주변 cell들의 값들을 참조하여 빈 cell의 값을 정하기 때문에 배열로 데이터를 관리하는 것이 연산을 수행함에 있어 효율적이다.

배열 데이터베이스 중에 가장 대표적인 SciDB를 이용하여 데이터를 관리하고, 분석 과정을 User Defined Operator (UDO)를 통해 구현하였다. 또한, 크게 두 가지 방향을 가지고 분석 과정을 최적화하였다. 첫 번째 방법은 chunk들 사이의 병렬 프로세싱을 활용하는 방법이다. SciDB는 배열을 chunk 단위로 나누어 관리를 하고 있으며, 여러 Instance들 안에 chunk들을 저장하고 있다[7]. Operator 수행 시, 각각의 Instance들 안에 있는 chunk들은 병렬적으로 수행된다. 예를 들어, 3개의 SciDB 노드 안에 각각 4개씩 Instance들이 있다면, 최대 12배의 병렬성을 가질 수 있는 것이다. 그래서 chunk size를 적절히 정의함으로써 SciDB 아키텍처 상에서 최대한의 병렬성을 활용하였다. SciDB의 shared nothing 구조를 통해 높은 scalability를 달성할 수 있었다. 두 번째 최적화 방법은 GPU를 활용하여 병렬 알고리즘을 구현하는 것이다. 배열 쿼리에 GPU를 사용하여 성능을 최적화하는 것은 적절한 시도이다[8, 9]. 대기 스캐닝 라이다 분석 과정에서 가장 많은 시간이 소요되고 반복되는 연산들이 많은 과정은 2D interpolation이며, 여러 방법들 중에 window interpolation을 GPU를 통해 최적화하였다.

본 논문에서 제안하는 방식을 통해 대기 스캐닝 라이다 데이터를 분석할 시에 기존에 RDBMS를 사용하는 방식과 Python을 통해 구현하는 방식보다 성능이 더 효율적임을 보였다. 또한, GPU를 통해 2D window interpolation을 수행하는 경우, CPU를 사용하는 경우보다 전체 분석 과정 시간에 있어 최대 9배의 성능 향상을 산출하였다. 본 연구의 Contribution은 다음과 같다.

(1) 대기 스캐닝 라이다 데이터를 배열 데이터베이스를 통해 관리하고 분석하는 방식을 제안하였다.

(2) 분석 과정을 UDO를 통해 구현하고, parallel chunk processing과 GPU를 활용한 병렬 알고리즘 두 가지를 통해 분석 성능을 개선하였다.

(3) 실제 시흥 및 김제에서 관측된 데이터를 토대로 미세먼지 농도를 분석하는 실험을 진행하였으며, 방대한 양의 데이터 분석 결과를 빠른 시간 안에 확인할 수 있음을 보였다.

본 논문은 본 저자가 참여한 [22] 논문에 기반을 두고 있다. 해당 논문의 parallel chunk processing 내용에 GPU를 추가적으로 적용하여 본 논문을 작성하였다.

제 2 절 논문의 구성

본 논문의 구성은 다음과 같다. 2장 관련 연구에서는 대기 스캐닝 라이다와 SciDB 관련된 연구를 소개한다. 3장에서는 UDO 구현과 최적화 방법에 대해서 제시한다. 4장에서는 실험 및 평가 내용을, 5장에서는 결론을 기술한다.

제 2 장 관련 연구

이번 장에서는 대기 스캐닝 라이다 관련 연구들과 SciDB 관련 연구들을 소개한다. 1절에서는 대기 스캐닝 라이다 분석 관련 내용들을 소개하고, 2절에서는 SciDB 아키텍처, User-Defined Operator, GPU를 활용한 배열 쿼리 가속화 사례에 대해 소개한다.

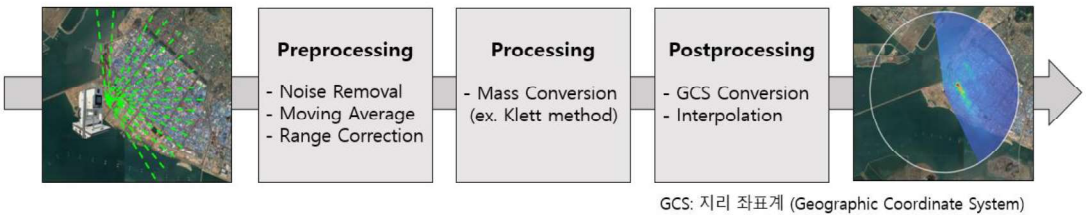
제 1 절 대기 스캐닝 라이다

대기 스캐닝 라이다는 특정 범위를 관측하여 해당 영역의 조밀한 대기 농도 결과를 얻어내는 용도로 사용된다. 이러한 결과를 얻기 위한 분석 과정은 일반적으로 [그림 1]과 같은 과정을 거치게 된다. 라이다 데이터를 사용하여 도시에서 환경 오염을 분석한 사례[3]에서는 RDBMS와 GIS를 활용하여 분석을 수행하였다. RDBMS를 통해 데이터를 관리하고, interpolation과 geospatial 분석 시에 GIS를 사용하였다. 해당 논문에서도 processing 전 후에 preprocessing과 postprocessing 과정을 정의하여 분석을 하고 있다.

대기 스캐닝 라이다 분석의 processing 과정에서는 일반적으로 Klett method[4]와 같이 기존에 과학자들이 연구한 농도 변환 식들을 이용하게 된다. 라이다 관측 상황이 달라지고, 수직 또는 수평 관측을 구분하면서 과학자들은 관측 파장을 다르게 적용하는 시도 등을 통해 정교한 알고리즘들을 연구하고 있다. [11]은 LiDAR 측정 데이터로부터 미세먼지 농도를 산출하는 알고리즘을 연구한 논문이다. 원리에 대한 간단한 설명은 다음과 같다. 특정 파장의 레이저를 대기 중으로 조사하면, 대기 중의 분자 및 미세먼지 입자 등과 부딪혀 산란현상을 발생시킨다. 대기 중에 발생한 산란 현상 중에서 일부는 역방향으로 후방산란 되어 라이다 관측 위치로 돌아오는데, 이를 망원경으로 수신하여 광신호를 검출한다. 이 때, 레이저가 발사된 후 수신되기까지 시간을 계산하여 신호를 분석하면, 거리 별 농도 분포를 산출할 수 있다.

여러 대의 라이다 장비를 사용하여 라이다 네트워크를 구성하고 유럽의 대기를 관측하려는 시도[5]를 통해서 이러한 대기 스캐닝 라이다가 향후에는 많은 지역을 cover하면서 빅 데이터를 생성할 것임을 예측할 수 있다. 그러므로 대기 스캐닝 라이다 빅 데이터를

파라미터를 변경해가면서 분석 결과를 확인하는 일들은 자주 발생하게 된다.



[그림 1] 대기 스캐닝 라이다 분석 과정

제 2 절 SciDB

SciDB[7]는 scientific 연구 분야에서 많이 사용되는 배열 데이터베이스이다. 천문학[12], 원격 센서[15, 16], 기후 모델링, 생물학[13] 등의 도메인에서 대용량 다차원 데이터의 관리 및 분석 용도로 사용된다. SS-DB[12]는 대표적인 배열 DBMS benchmark이며, Large Synoptic Survey Telescope (LSST)와 같은 천문학 관측 application을 가지고 image manipulation 등의 연산들을 정의하여 MySQL과 SciDB 성능을 비교하였다. 해당 비교 결과를 통해 RDBMS는 scientific data를 처리하기에 적합하지 않은 반면, SciDB는 여러 아키텍처 특성을 활용해서 더 좋은 성능을 가져올 수 있음을 보였다. GenBase[13]는 genomics 도메인에서 SVD, biclustering, covariance, predictive modeling과 같은 복잡한 연산들을 정의하여 여러 데이터베이스에서 성능을 비교하였다. Postgres, Hadoop, SciDB 등을 비교하였으며, SciDB가 위의 analytic queries를 잘 처리함을 보였다. EarthDB[12]는 NASA MODIS 데이터를 SciDB를 통해 관리하고 분석하는 usecase를 보여주었다. 또한, 이를 visualization하는데 있어서 dynamic prefetching 아이디어를 제시한 사례[16]도 존재한다. 위의 연구들을 통해 SciDB가 scientific data 분석에 적합하며 많이 사용되고 있음을 알 수 있다. SciDB 이외에 배열 데이터베이스에는 RasDaMan[17], TileDB[18], ChronosDB[19] 등이 있다.

SciDB가 RDBMS에 비해 scientific data를 processing하기에 적합한 이유는 다음과 같다. SciDB[7]의 array 모델은 index 없이도 데이터 attribute에 대한 direct 접근이 dimension을 통하여 가능하다.

SciDB는 column-oriented design을 하고 있으며, attribute들이 따로 저장이 된다. 같은 쿼리에 대해서 SciDB는 chunk size를 적절히 정의함으로써 parallel processing이 가능할 수 있지만, RDBMS는 일반적으로 다른 chunk를 함께 봐야 할 수 있기 때문에 sequential processing을 요구할 수 있다. RDBMS의 table 모델과 비교하여 SciDB의 array 모델이 갖는 이점은 [그림 2]에 나오는 subarray query 예시에서 찾아볼 수 있다. Array 모델이 인접한 cell 사이의 locality를 더 잘 보존하고 있다. 또한, table의 경우에 dimension 정보까지 column으로 두어 record로 저장을 하고 있어야 한다. 그에 반해, array는 attribute 정보만 저장을 하면 된다.

dim1	dim2	attr1
0	0	10
0	1	20
0	2	15
0	3	18
1	0	30
1	1	21
1	2	22
1	3	31
2	0	20
2	1	25
2	2	23
2	3	34
3	0	43
3	1	22
3	2	26
3	3	13

< table >

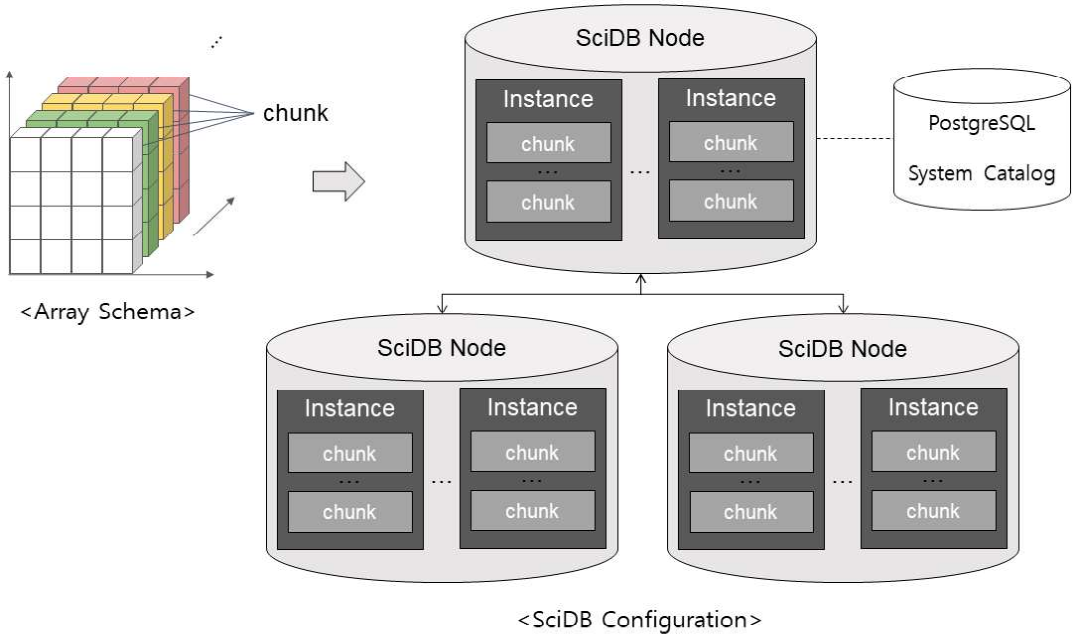
10	20	15	18
30	21	22	31
20	25	23	34
43	22	26	13

< array >

[그림 2] Table과 Array에서의 Subarray Query 비교

1. 아키텍처

SciDB는 배열을 기본 데이터 모델로 하고 있으며, 배열을 chunk 단위로 나누어 관리를 한다. 이러한 chunk들은 Instance안에 분배되어 저장이 되고, 하나의 노드 안에 여러 Instance들이 존재하게 된다. SciDB는 shared nothing 구조를 가지고 있으며, distributed computing은 Map/Reduce[6] model에 영향을 받았다. 만일 3개의 SciDB 노드 안에 4개씩 Instance들이 있다면, 최대 약 12배의 병렬성을 가질 수 있다. [그림 3]는 SciDB 아키텍처의 예시를 보여주고 있다. 하나의 master 노드와 두 개의 slave 노드로 구성된다. PostgreSQL은 system catalog의 역할을 하며, array metadata와 같은 정보들이 table 형태로 저장되어 있다. 3차원 배열이 chunk 단위로 나뉘어 3개의 노드에 저장되는 것을 보여주고 있다. Chunk size는 임의대로 지정할 수 있으며, [그림 3]에서는 두 가지 차원에 대해서는 해당 차원의 전체 크기로 가져가고, 나머지 한 차원은 1로 가져간 형태이다.



[그림 3] SciDB 아키텍처 예시

2. User-Defined Operator (UDO)

UDO는 사용자가 구현을 한 operator로써 build하여 만들어진 library 파일을 SciDB에 load하여 사용할 수 있다. 이러한 UDO는 SciDB에서 built-in operator로서 동작이 된다. Scientific algorithm들을 UDO를 통해 구현을 하여 수행[9]한 경우에 성능 개선의 기회가 많아지게 된다. 이는 데이터베이스와 외부 알고리즘 사이에 데이터 이동을 피할 수 있고, 데이터베이스에서 제공하는 optimal한 데이터 관리 방법을 사용할 수 있기 때문이다. 예를 들어, SciDB에서 제공하는 Instance들 사이의 parallel한 프로세싱을 통해 성능 개선을 할 수 있게 된다. 이 밖에도 UDO를 활용한 사례들로는 배열로 표현된 binary mask의 connected components labeling 구현[14], SciDB의 window operator를 Incremental 한 방식을 통해 최적화한 경우[21], Progressive Top-k subarray query processing을 배열 데이터베이스 안에서 구현을 한 경우[20] 등이 존재한다. 이와 같이 UDO를 통한 구현은 복잡한 분석을 SciDB 안에서 가능하게 하며, 많은 성능 개선의 기회들이 주어진다.

SciDB의 쿼리 language로는 Array Query Language (AQL)과 Array Functional Language (AFL) 두 가지가 존재한다. AQL은 high-level declarative language로 관계형 데이터베이스의 SQL과 유사하며, 동작 시에 AFL로 변환되어 수행이 된다. UDO는 AFL 형태로 load가 되어 동작이 된다.

3. GPU를 활용한 배열 쿼리 가속화

GPGPU는 General-Purpose computing on Graphics Processing Units를 의미하며, 이를 통해 기존에 CPU가 하던 응용 프로그램들의 계산들을 수행할 수 있게 되었다. GPU는 여러 연산들을 동시에 처리할 수 있도록 CPU보다 더 많은 core와 Arithmetic Logic Unit (ALU)가 존재한다.

CUDA는 Compute Unified Device Architecture의 약자로 Nvidia에서 개발하고 있으며, GPU를 사용한 병렬 알고리즘을 C언어 등을 통해 작성할 수 있는 기술이다. CUDA의 processing flow는 크게 세 가지 과정이 순서대로 수행이 된다. 첫 번째는, 메인 메모리에서 GPU의 메모리로 처리할 데이터들을 복사한다. 두 번째는, GPU 상에서 병렬적으로 처리를 하게 되며, 마지막 세 번째는 결과를 다시 메인 메모리로 복사를 하게 된다. 두 번째 과정이 Kernel이 수행되는 과정이며, Kernel은 병렬적으로 수행되는 함수이다. CUDA GPU의 구조는 Grid, Block, Thread 총 3가지로 구성된다. Grid, Block, Thread 순서대로 뒤에 오는 것이 더 작은 구조이다. 즉, 여러 Thread가 모여 Block이 되고, 여러 Block이 모여 Grid가 된다. Grid는 Device를 의미하고, Block은 Streaming Multiprocessor (SM)을 의미하며, Thread는 CUDA core를 의미한다. Kernel 함수를 정의하고 호출할 때, Block 개수와 Thread 개수를 상황에 맞게 적절하게 설정해야 한다. SM은 기본적으로 32개의 Thread를 Warp 단위로 정의하고 있으며, 실행 시 Warp 단위로 수행이 된다. CUDA는 SIMT (Single Instruction Multiple Thread)를 추구하며, 하나의 명령어로 여러 개의 스레드를 동작시키는 것을 목표로 한다.

배열 쿼리에 GPU를 사용하여 성능을 높인 사례들이 존재한다. 이때 발생하는 문제점과 해결방안, GPU 사용의 장단점 등을 분석한 연구[8]에서는 Dense Matrix Multiplication, K-means clustering, PageRank, Image smoothing 총 4가지 application들을 SciDB 내부에서 GPU를 사용한 쿼리 형태로 수행하였다. 배열 쿼리에 GPU를 적용하는 것은 적절한 시도이며, CPU와 GPU 각각에 overhead가 많이 생기지 않도록 일을 분배하는 것이 필요하다. CPU는 Kernel execution에 overhead가 있으며, GPU는 data transfer에 overhead가 있으니, 일을 적절히 분배하여 해소하는 것이 필요하다. 기존의 CPU 버전의 Differential Emission Measure (DEM) operator 구현[9]에

GPU를 적용한 연구[10]에서는 GPU overhead를 줄이기 위해 여러 시도를 하였다. Concurrent asynchronous 수행과 Interleaved work를 통해 GPU를 최대한으로 활용하고, 성능을 개선하였다. 다양한 실험들을 통해 GPU의 사용이 과학자들로 하여금 짧은 시간 안에 분석 결과를 확인할 수 있게 해준다는 것을 확인하였다.

제 3 장 대기 스캐닝 라이다 분석 최적화

3장에서는 대기 스캐닝 라이다 분석 과정과 최적화 방법들에 대해서 기술한다. 1절에서는 분석 과정들에 대해서 설명하고, 구현된 함수들과 UDO에 대해서 자세히 기술한다. 2절에서는 최적화 방법들로 SciDB 아키텍처 상의 병렬 프로세싱과 GPU를 활용한 병렬 알고리즘에 대해서 서술한다.

제 1 절 분석 과정

1. 함수 및 파라미터

대기 스캐닝 라이다 분석 과정은 크게 3가지 preprocessing, processing, postprocessing로 구성된다. 이 과정들을 UDO로 만들기 위해 구현한 대표적인 함수들과 사용되는 파라미터 리스트는 [표 1]과 같다. 함수들을 특성에 따라 분류해 봤을 때, preprocessing과 processing, geo-spatial, interpolation으로 나눌 수 있다. Interpolation 같은 경우에 여러 방법들이 구현되어 있는데, 이는 사용자가 필요에 따라 그에 맞는 함수를 사용할 수 있게 하기 위함이다. 파라미터의 경우에 여러 옵션들이 구현되어 있으며, 과학자들이 파라미터가 변화함에 따라 분석 결과가 어떻게 달라지는지 확인하고 싶을 때 사용할 수 있다.

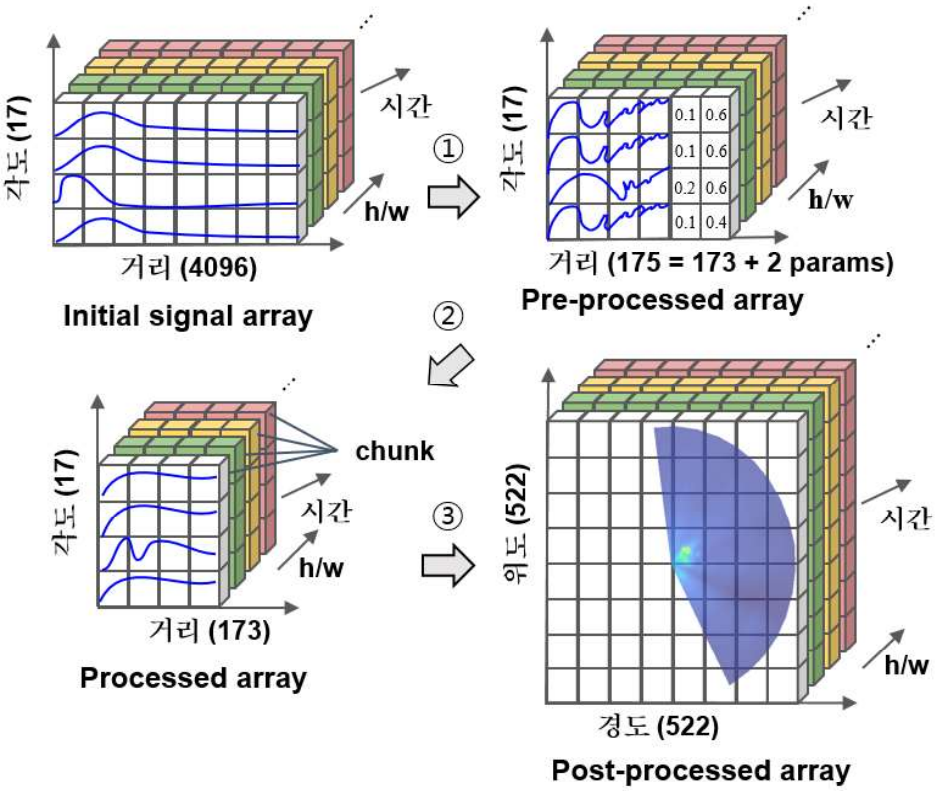
함수	**Preprocessing & Processing**	Moving Average / Range Correction / Noise Reduction / Window Summation / Integration by Parts
	Geo-spatial	Coordinates Conversion
	Interpolation	Bilinear / Bicubic / Delaunay / Linear / Window
파라미터	Window Size / Interpolation Array Size / Reference Distance / Reference Mass / The Number of Background Noises	

[표 1] 대표적인 함수와 파라미터

2. UDO 구현

분석 과정에 맞게 Preprocess (①), Process (②), Postprocess (③) 총 3개의 UDO를 구현하였다. 구현함에 있어 세부 내용은 실제 시흥에서 진행된 미세먼지 관측에서의 분석 과정[11]에 기반을 두고 있다. 미세먼지 분석이 진행됨에 따라 배열의 스키마가 변해가는 과정은 [그림 4]과 같다. 실제 시흥에서는 h/w 하나를 가지고 테스트를 하였지만, 해당 그림은 여러 대로 확장된 시나리오를 표현하고 있다. 모든 배열들은 총 4개의 차원들로 구성되어 있으며, chunk들은 다른 색깔로 구분을 하였다. 즉, Initial signal array의 chunk size는 17 X 4096이다. 괄호 안에 있는 숫자는 해당 dimension의 크기이다. 예를 들어, 각도는 17개로 구성되어 있다. 분석 과정에서의 자세한 내용은 아래 각각의 UDO를 설명하는 부분에서 다루고 있다.

UDO 구현 시, chunk 단위로 데이터를 읽어와서 어떻게 수행을 할지를 [표 1]에 있는 함수들을 통해 구현하였고, 해당 함수를 통해 처리된 내용을 chunk 단위로 저장을 하게 된다. 또한, Input과 Output 배열의 schema를 [그림 4]에 나와 있는 형태로 정의하였으며, 해당 schema에 맞게 데이터를 읽고 처리하여 저장을 한다.



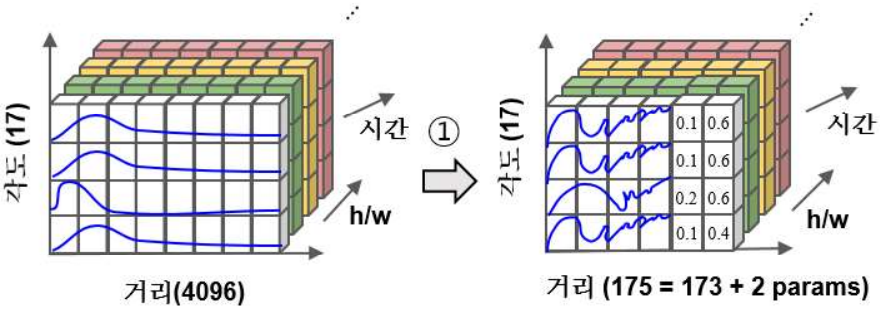
[그림 4] 확장된 시흥 미세먼지 분석 과정

2.1 Preprocess

Preprocess (①) 과정의 Input 배열과 Output 배열은 [그림 5]에 나타나 있다. Input 배열의 chunk size는 17 x 4096이고, 이 때의 거리 간격은 4.8m이다. 즉, 17개의 각도에 대해서 20km까지 관측된 라이다 Raw data가 입력으로 들어온다. Output 배열의 chunk size는 17 x 175이며, 이 때의 거리 간격은 28.8m이다. 28.8m 간격이라면 거리의 point 개수가 175개가 아닌 682개 맞지만, 거리가 멀어질수록 분석 결과에 대한 신뢰도가 낮아 질 수 있기 때문에 5km까지 분석 결과를 제한한다.

일반적으로는 모든 각도의 4096개 point들이 독립적으로 똑같은 전처리 과정을 거치게 되지만, 만일 Reference data를 사용하게 되는 경우, 특정 각도의 데이터들을 먼저 연산 해주는 경우가 발생할 수 있다. 예를 들어, 시흥에서는 두 지점의 관측 데이터를 참조하였고, 특정 두 각도를 먼저 계산하고 나서 구해지는 특정 상수를 가지고, 나머지 15개의 각도에 대해서 처리를 하였다. Preprocess 과정 가운데 각도별로 각각 2개의 파라미터가 생성되는데, 이 정보는 Process 과정에도 사용이 되므로 전달을 해주어야 한다. 이를 위해 해당 파리미터 값들을 거리의 마지막 위치에 기록해둔다. 그래서 5km까지라면 173개의 거리가 맞지만, 2개의 파리미터를 더 기록하기 때문에 175개가 결과로 나오게 된다. [그림 5]에서 output 배열의 cell 안에 적혀 있는 숫자들은 2개의 파라미터의 예시 값들을 적어놓은 것이다. 해당 파라미터의 의미와 계산하는 과정에 대한 상세한 설명은 Process 과정에서 설명을 하고 있다. 한 각도에서 4096개의 전 처리 순서는 다음과 같다.

1) 6개씩 합산: 28.8m 간격의 682개 points로 변경됨.
2) Noise 제거: 뒤에 200개의 평균을 모든 point들에 감산.
3) Moving Average: window size는 5로 한다.
4) Range correction: 시작 지점으로부터 거리의 제곱을 곱함.
5) 모든 point들에 대해 log 값을 계산한다.



[그림 5] Preprocess 과정의 Input과 Output

2.2 Process

Process (②)의 Input과 Output은 [그림 6]에 나타나 있다. Input, output 배열의 chunk size는 각각 17 x 175, 17 x 173이고, 이 때의 거리 간격은 모두 28.8m이다. 모든 각도의 173개 point들에 대해서 Klett Method[4]의 식(14)과 뒤에 2개의 파리미터를 이용하여 5km 지점까지의 소산 계수[①]를 계산한다. 해당 식은 다음과 같다.

$$\sigma\,(r) = \frac{exp[(S - S_m)/k]}{\sigma_m^{-1} + \frac{2}{k}\int_r^{r_m} exp[(S - S_m)/k]dr'}$$

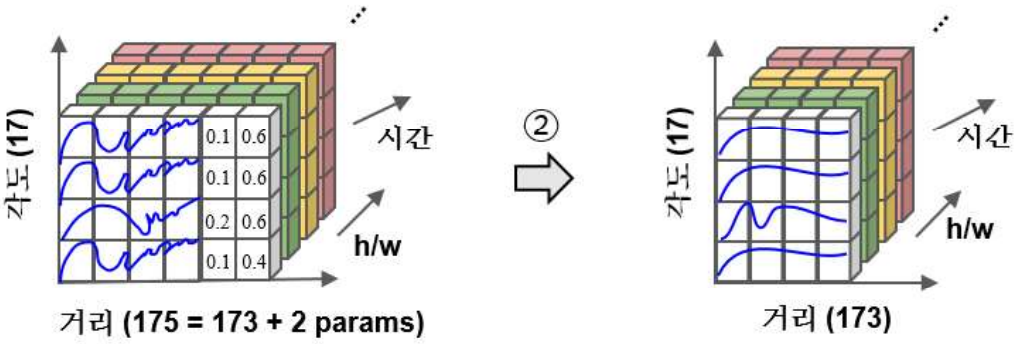
r_m 은 reference 지점으로 5km를 의미하며, S_m 은 reference 지점에서의 신호 세기이다. K는 상수 값으로 0.7을 사용하였으며, σ_m [②]은 reference 지점에서 앞으로 500m, 뒤로 500m 구간에서의 기울기 값이다. 적분 계산 시 reference 지점에서의 값을 0으로 잡고, backward 방향으로 거리를 옮겨가며 계산되는 $exp[(S - S_m)/k]$ 값을 누적하여 계산한다. 위 식의 결과를 소산 효율 값 (ex. 8)으로 나누어 PM10을 계산하고, PM10 값에서 지점 관측 농도의 비율[③]을 고려하여 PM2.5의 농도 값을 계산한다.

Preprocess 과정에서 전달되는 두 가지 파라미터에 대한 자세한 설명은 다음과 같다. 첫 번째 파라미터 σ_m 값은 특정 위치의 지점 관측 농도 값과 일치하게 나오게 하는 σ_m 값을 Klett method의 식(14)를 이용해 역으로 계산을 한다. 그렇게 계산된 두 각도에서의 σ_m 값을 가지고 linear interpolation을 통해 나머지 각도에서의 σ_m 값을 계산한다. 두 각도에서의 σ_m 값의 min, max 값을 가지고, interpolation한 값이 min보다 작으면 min값으로, max값보다 크면 max값으로 하여 최대 최소 값을 제한하였다. 두 번째 파라미터인 지점 관측 농도의 비율 값은 두 지점의 PM10과 PM2.5의 비율 값을 가지고 첫 번째 파라미터의 interpolation 방식과 동일한 과정을 거쳐서 계산을 하였다. 즉, 두 각도에서는 실제 비율 값이 적용이 되고, 나머지

---

① 소산 계수는 대기 중에 있는 분자와 미세먼지 입자 등과 부딪혀 빛이 감쇄하는 정도를 의미한다.
② 해당 σ_m 값이 Preprocess 과정에서 전달된 첫 번째 파라미터이다.
③ 해당 지점 관측 농도의 비율 값이 Preprocess 과정에서 전달된 두 번째 파라미터이다.

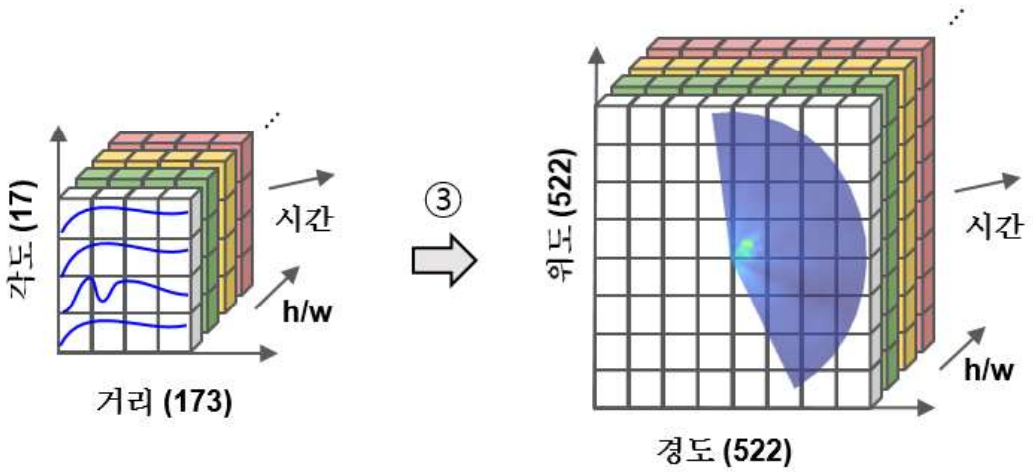
각도들에서는 interpolation한 값들이 적용이 된다.



[그림 6] Process 과정의 Input과 Output

2.3 Postprocess

Postprocess (③)의 Input과 Output 배열은 [그림 7]에 나타나 있다. Input 배열의 chunk size는 각각 17 x 173이다. Output 배열의 chunk size는 522 x 522이다. 또한, 기존의 각도와 거리 축들이 위도와 경도 축으로 변경된다. 즉, 축을 지리 좌표계에 맞도록 변환을 하고, Interpolation을 통해 dense한 output을 만드는 과정이다. 모든 각도의 173개 point들에 대해서 위도, 경도 인덱스에 알맞은 위치에 넣어준다. 이 때, 522는 시각화 resolution과 배열의 적절한 크기를 고려한 값이다. 그 다음에 Interpolation을 통해 5km 반경의 관측 범위 안에 들어오면서, 비어있는 cell들에 대한 값들을 채워준다. Interpolation 알고리즘은 여러 방법이 존재하는데, delaunay triangulation[④]을 이용한 방법을 사용하였다. 비어있는 cell에 대하여 이미 채워진 값들로 이루어진 가장 가까운 triangle을 찾아 농도 값을 구해내는 것이다.



[그림 7] Postprocess 과정의 Input과 Output

④ C++ library: http://rncarpio.github.io/delaunay_linterp/

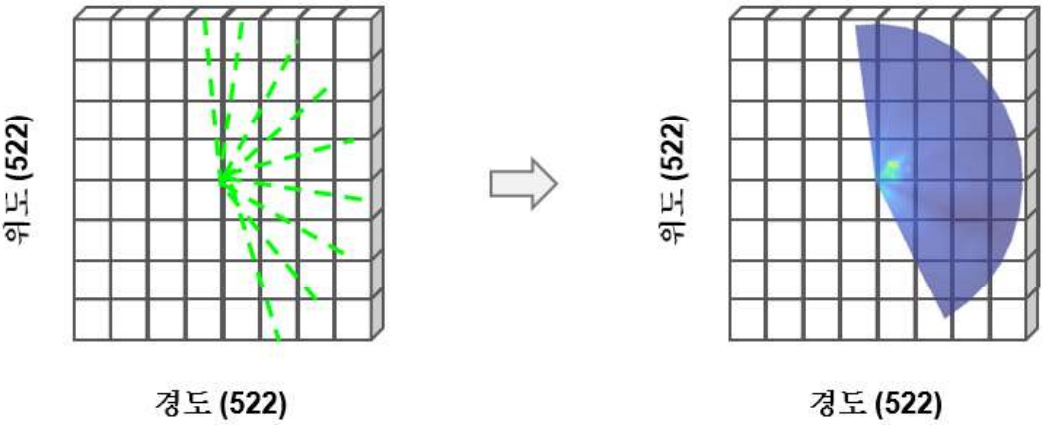
제 2 절 분석 최적화

1. SciDB 아키텍처 상의 병렬 프로세싱

구현된 UDO를 수행하게 되면, 각 Node의 Instance들 안에 있는 chunk들은 parallel하게 처리된다. Chunk size를 정함에 있어 독립적인 dimension에 대해서는 해당 dimension의 size를 1로 하고, 비독립적인 dimension에 대해서는 해당 dimension의 size를 dimension의 전체 크기로 가져간다. 예를 들어, [그림 4]의 Initial signal array에서 chunk size는 시간 (1) x h/w (1) x 각도 (17) x 거리 (4096)의 크기를 갖는다. Reference data를 활용할 때, 특정 각도를 먼저 계산해야 하기 때문에 각도 dimension은 비독립적이다. [그림 4]에서 chunk size는 계속 변화하지만 h/w와 시간상의 병렬성은 계속 유지된다.

2. GPU를 활용한 병렬 알고리즘

전체 분석 과정 중에 가장 많은 시간이 소요되며, 반복되는 연산들이 많은 부분은 Postprocess 과정의 interpolation이다. 이는 각 cell 마다 수행 과정에서 병렬성이 뚜렷하기 때문에 GPU를 통한 성능 최적화가 적절하다. 여러 interpolation 알고리즘들 중에 Window를 이용한 방식의 최적화를 시도하였다. 2D window interpolation은 delaunay triangulation 방식과 유사한 결과를 보이며, GPU를 사용하여 병렬화 하기에 용이하다. 과정은 다음과 같다. 관측 범위 안의 비어있는 cell들에 대하여 window aggregation을 통해 주변의 실제 값들로 채워진 cell들의 평균 값으로 채워준다. Window size는 3에서 시작하여 1씩 증가시키며 최소 3개 이상의 채워진 cell들을 찾으면 평균을 계산한다. 시간 복잡도는 $O(c \cdot w \cdot n^2)$이다. (n x n chunk: n = 522, w = 3 x 3에서 시작하여 1씩 증가, c: chunk 개수) 즉, [그림 8]에서 하나의 chunk를 기준으로 할 때, 522 x 522 배열을 대상으로 각 cell들의 연산들이 GPU를 통해 병렬적으로 처리된다. Kernel 함수 호출 시 Block와 Thread 개수는 모두 522개로 하였다.



[그림 8] Interpolation 예시

제 4 장 실험 및 평가

4장에서는 대기 스캐닝 라이다 분석의 실험 결과와 최적화 성능 평가에 관해 서술한다. 1절에서는 parallel chunk processing에 관한 실험, 2절에서는 GPU를 활용한 실험에 대해서 각각의 실험 환경과 결과에 대한 평가 및 분석을 다루고 있다.

제 1 절 Parallel Chunk Processing

본 실험에서는 3개의 워크스테이션을 사용하였다. 각각의 워크스테이션 환경은 Ubuntu 16.04.6 LTS, i7-4790S CPU, 8GB memory이며 4개의 instance를 가지는 SciDB 19.11이 설치되어 있다. Python 3.5, Numpy 1.11, PostgreSQL 13.2을 사용하였다. 모든 UDO는 C++로 구현하였다. 본 실험은 시흥에서 실제로 진행된 미세먼지 관측 시나리오를 확장하여 설계하였다.

비교군은 크게 두 가지를 설정하였다. 첫 번째는, 과학자들이 일반적으로 많이 사용하는 Python과 Numpy를 이용한 구현을 비교하였다. 두 번째는, 이전 연구들에서 많이 사용했던 RDBMS를 이용한 방식을 비교하였다. 저자가 조사한 바에 따르면, 본 논문이 Array-based DBMS를 대기 스캐닝 라이다 분석에 적용한 첫 사례이기 때문에, RDBMS를 비교군으로 잡는 것은 불가피한 선택이었다.

실험 결과는 [표 2]에서 보여주고 있다. Task는 [그림 4]에서 언급된 Preprocess (①), Process (②), Postprocess (③) 세 가지 과정으로 구성된다. ①+②+③은 전체 signal processing을 의미하며, interpolation은 delaunay 알고리즘을 사용하였다. ①+②로 구성된 경우에는 PostgreSQL을 사용하는 경우 interpolation을 내부 쿼리에서 구현할 수 있는 적절한 방법이 없었기 때문에 ③ 과정을 제외시켰다. Data는 3개의 dimension h/w, 각도, 거리로 구성된다. 이들 중에 각도, 거리 크기는 각각 17개, 4096개로 고정시켰으며, h/w size는 4개 또는 1200개를 사용하였다. [표 2]에 나와 있는 Data 크기는 h/w size를 의미한다. 실험에서 사용되는 모든 Dataset은 시흥에서 관측된 실제 데이터를 가지고 synthetic 하게 만들어 사용하였다. Node는 단일 노드 또는 3개의 노드 두 가지 환경에서 실험을 하였다. PostgreSQL은 h/w

개수를 4등분하여 300개씩 parallel 하게 쿼리를 실행하였고, 이를 통해 SciDB의 4개 instance와 비슷한 효과를 얻게 하였다. PostgreSQL의 분산 환경은 postgres_fdw를 이용하여 구현하였다. 실험은 3가지로 나누어 진행이 되었으며, 각각의 결과에 대한 분석은 아래와 같다.

Task	Data	Node	Comparison	Time (sec)
①+ ②+ ③	4	1	Python	86.16
			SciDB	**19.63**
①+ ②	1200	1	PostgreSQL	551.91
			SciDB	**3.43**
①+ ②	1200	3	PostgreSQL	1018.59
			SciDB	**1.17**

[표 2] Chunk Processing 실험 결과

1. Python vs SciDB UDO

Python으로 구현한 것과 SciDB UDO로 구현했을 때 결과를 비교해보면, SciDB UDO가 instance 각각의 parallel processing으로 더 빠른 결과를 얻을 수 있었다. SciDB Node 하나에 총 4개의 Instance가 존재하므로, 약 4배 정도의 속도 개선을 확인하였다.

2. PostgreSQL vs SciDB UDO (Single Node)

Single Node에서 PostgreSQL과 SciDB UDO를 비교했을 때, SciDB UDO가 더 좋은 성능을 보였다. 이는 PostgreSQL에서는 analytic query를 수행함에 있어 빈번한 scan이 발생했기 때문이다. 즉, RDBMS에서 analytic query의 사용은 적절하지 않음을 보여주고 있다. 또한, 기본적으로 데이터 모델의 차이 (table vs array)에서 성능의 차이가 발생하였다. Multidimensional array를 chunk 단위로 나누어 관리하는 방식과 dimension 정보를 table에 column으로 관리하는 방식은 특정 dimension에 접근할 때 속도 차이가 발생하며, 결국 array 모델이 locality를 잘 보존한다고 볼 수 있다.

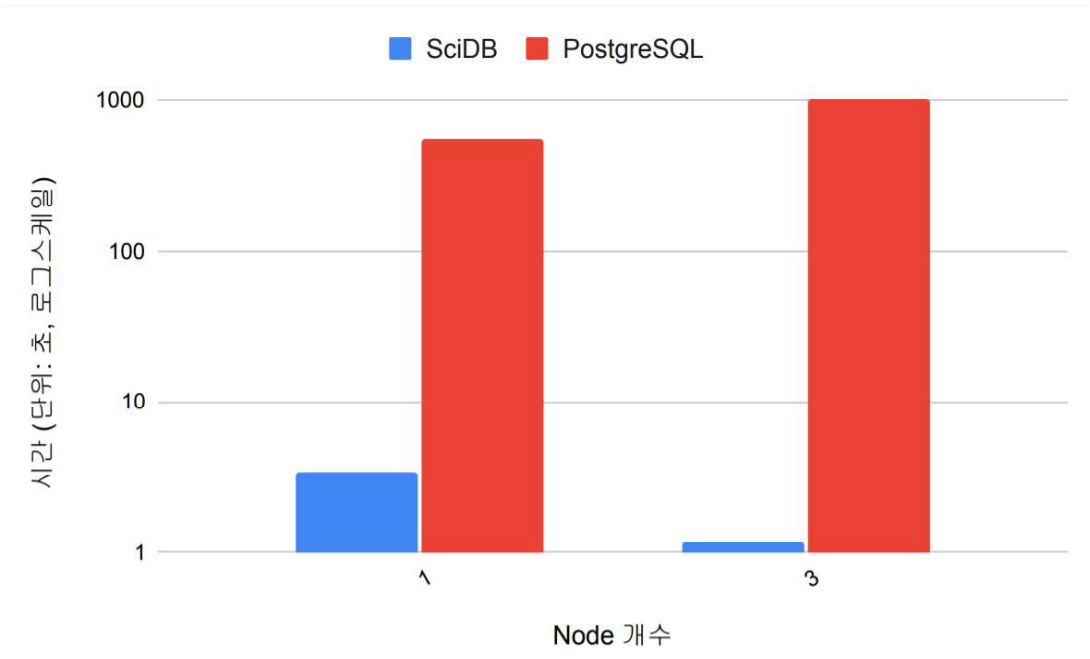
PostgreSQL Query의 explain 결과를 분석한 내용은 다음과 같다. 자주 사용되는 연산에는 subquery scan, sequential scan, external merge sort, quick sort, hash aggregate, window aggregate, merge join,

hash join 등이 사용되었다. 이들 연산들 중에 scan과 aggregate 연산들이 상대적으로 많은 시간을 차지하였다. 또한, 전체 수행 시간 중에 가장 많은 시간을 소요한 critical한 부분은 hw_idx 1200대를 for loop을 수행하면서 reference data를 참조하는 부분이다. hw 개수가 1대일때는 1초 미만의 시간이 걸렸지만, 1200대가 되면서 수행 시간이 해당 배수만큼 증가하여 느린 결과를 보이고 있다.

3. PostgreSQL vs SciDB UDO (Multi Nodes)

[그림 9]은 [표 2]의 Scalability에 대한 실험 결과를 그래프 형태로 표현한 것이다. X축은 Node 개수이며, Y축은 소요된 시간을 로그스케일로 나타낸 것이다. 그러므로 노드 개수가 커졌을 때, Y축의 값이 작아질수록 scalable 하다고 볼 수 있다. 정확한 시간 수치는 [표 2]에 기록되어 있다. SciDB는 multi node에서 single node보다 더 좋은 성능을 보인 반면에, PostgreSQL은 single node가 더 좋은 성능을 보였다. 이는 PostgreSQL이 Multi node 환경에 적합하지 않음을 보여주고 있다. 여러 노드들 사이에 데이터 전달이 network overhead를 발생시키며 complexity가 높아지게 된다. 또한, work가 한 노드에 쏠리게 되어 성능의 저하가 발생하는 것으로 확인되었다. 이러한 문제들을 봤을 때, postgres_fdw를 통한 분산 환경의 구현에 한계가 있는 것으로 볼 수 있다.

PostgreSQL Query의 explain 결과를 분석한 내용은 다음과 같다. 대부분의 처리 시간 비율은 single node에서의 결과와 유사하다. 다만 달라진 점은 scan 결과를 다른 노드에서 가져오고, 다시 다른 노드에 insert할 때 발생한 network cost가 많은 시간을 차지하였다. 또한, parallel processing이 이루어지지 않고, 한 노드에서 sequential processing이 발생하여 single node보다 더 안 좋은 성능을 보이게 되었다.



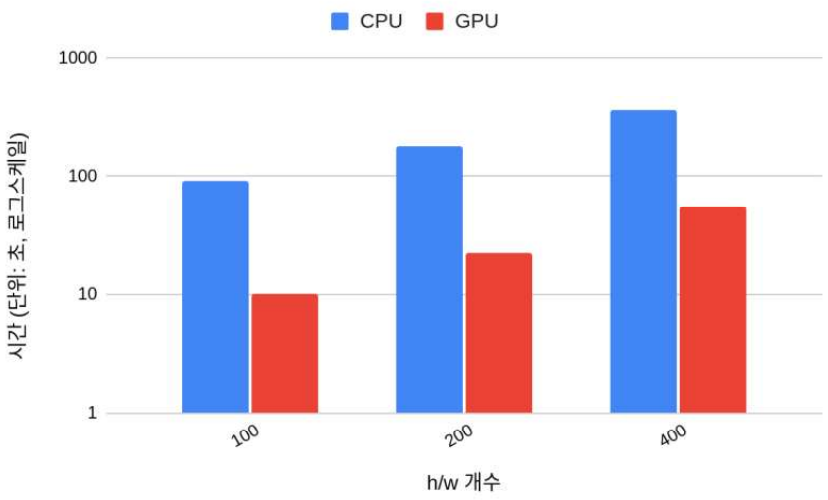
[그림 9] Scalability 실험 결과

제 2 절 GPU 가속화

본 실험에서는 1개의 워크스테이션을 사용하였다. 워크스테이션은 Ubuntu 18.04.3 LTS, AMD Ryzen 7 1700, GeForce RTX 2080 SUPER, 64GB memory의 환경을 가지고 있다. SciDB는 Ubuntu 16.04 이하 버전에서만 설치가 가능하므로, nvidia/cuda:11.0-devel-ubuntu16.04 docker image를 사용하였다. 해당 image에는 Ubuntu 16.04.7 LTS가 설치되어 있으며, 4개의 instance를 가지는 SciDB 19.11를 추가적으로 설치하였다. GPU 프로그래밍 언어로는 CUDA C 11.0을 사용하였다. GPU 실험에서는 모두 window interpolation 방식을 사용하였다. Data는 4개의 dimension h/w, 시간, 각도, 거리로 구성된다. 이들 중에 시간, 각도, 거리 크기는 각각 3개, 36개, 4096개로 고정시켰으며, h/w size는 100개, 200개, 400개 중 하나를 사용하였다.

1. CPU vs GPU 전체 분석 시간

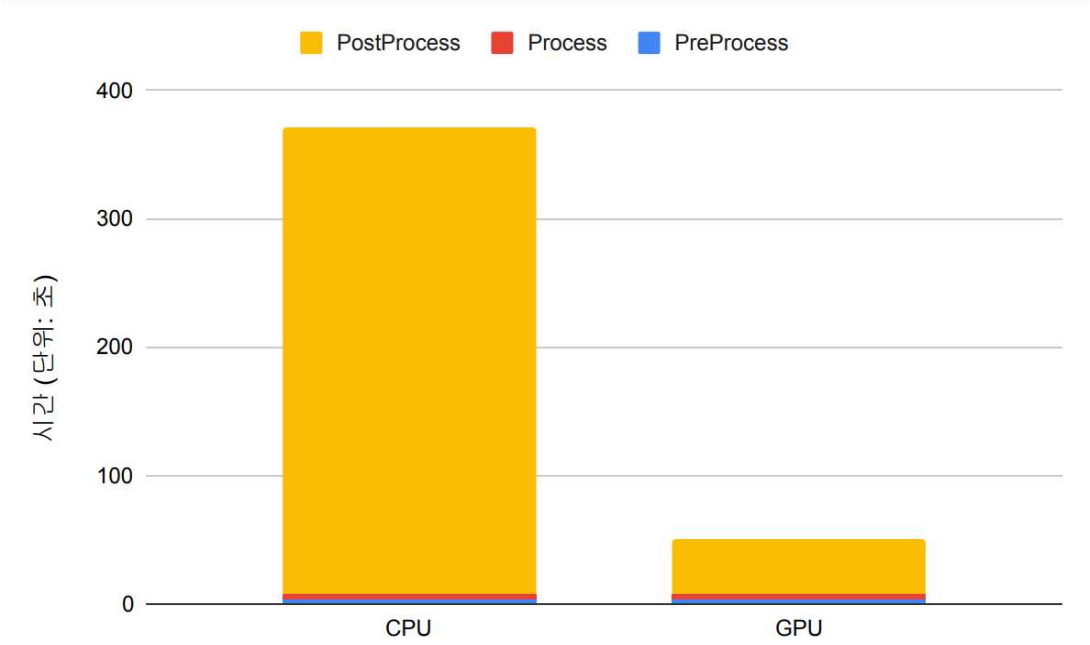
CPU와 GPU를 전체 분석 시간을 통해 비교한 실험 결과는 [그림 10]에서 보여주고 있다. 분석 과정은 전체 signal processing의 분석 시간을 비교하였으며, Preprocess, Process, Postprocess를 모두 포함한 시간이다. [그림 10]의 X축은 h/w 개수이며, Y축은 초 단위의 시간을 로그 스케일로 나타낸 것이다. GPU를 사용했을 때, CPU 대비 최대 9배의 속도 개선을 산출하였다.



[그림 10] GPU 가속화 실험 결과

2. UDO 별 분석 시간

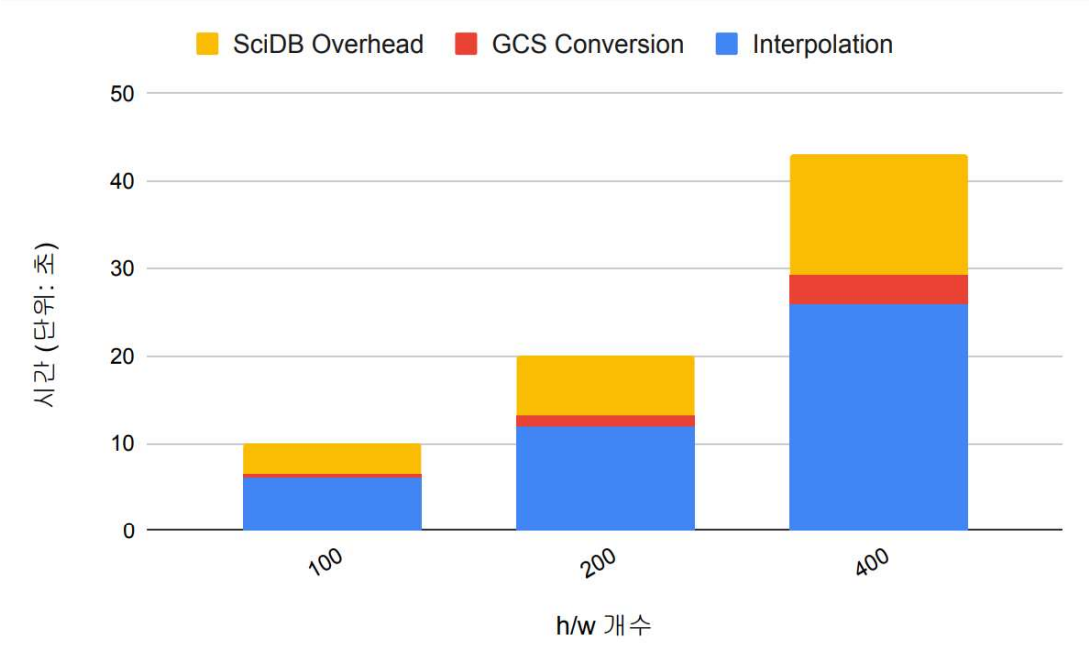
[그림 11]은 h/w 개수가 400개일 때 CPU와 GPU 각각의 Preprocess, Process, Postprocess 별 소요시간을 보여주고 있다. 3개의 UDO의 소요시간을 봤을 때, Postprocess과정이 나머지 두 개의 operator에 비해 압도적으로 많은 시간을 차지하였다. Postprocess 과정에서만 GPU를 사용한 것이기 때문에, Preprocess와 Process 과정은 CPU와 GPU 모두 동일한 결과를 보였다. Preprocess와 Process 모두 각각 4초정도의 시간이 소요되었다. 그에 비해, Postprocess에서는 CPU가 364초, GPU가 43초 정도 소요되었다.



[그림 11] UDO 별 소요 시간

3. Postprocess 세부 분석 시간

[그림 12]은 h/w 개수를 100, 200, 400개 증가시킬 때 GPU를 통해 최적화된 Postprocess operator의 세부 소요시간을 보여주고 있다. SciDB Overhead는 데이터 read and write 시간을 의미한다. GCS (Geographic Coordinate System) Conversion은 지리 좌표계로 변환하는 작업을 의미한다. Interpolation은 GPU로 최적화된 부분으로써 cudaMalloc, cudaMemcpy를 통한 data transfer, kernel 실행 과정을 모두 포함한 측정 시간이다. SciDB Overhead, GCS Conversion, Interpolation 3가지의 소요 시간 비율은 평균적으로 5:1:10 정도를 나타내었다.



[그림 12] GPU 버전의 Postprocess 소요 시간

제 5 장 결 론

본 논문에서는 대기 스캐닝 라이다 데이터를 배열 기반 데이터베이스 중 하나인 SciDB에 저장하고, 분석 과정을 내부의 User Defined Operator를 통해 구현함으로써 분석 성능을 개선할 수 있음을 보였다. 과학자들이 파라미터를 변경해 가며 분석 결과를 확인하는 작업은 빈번하게 발생하기 때문에 데이터 크기가 증가했을 때 분석 성능을 빠르게 하는 것이 필요하다. 구체적인 최적화 방법으로는 SciDB 아키텍처 상의 parallel processing과 GPU를 활용한 분석 알고리즘 병렬화 두 가지를 사용하였다. 실험을 통해 기존에 RDBMS analytic query 또는 Python을 이용한 구현보다 SciDB UDO를 이용한 방식이 효과적임을 보였으며, GPU를 이용했을 때 UDO의 성능 개선 또한 확인하였다.

이를 통해 과학자들은 짧은 시간 안에 크기가 큰 데이터의 분석 결과를 얻을 수 있다. 또한, 실제 시흥과 김제에서 관측된 데이터를 토대로 미세먼지 농도를 계산하는 과정을 구현하였기에 현실에서 바로 적용이 가능하며, 미세먼지 이외에 다른 대기 분석에도 일반화하여 적용이 가능하다. 본 논문의 시나리오는 배열 데이터베이스를 활용한 application 또는 usecase에 해당한다.

후속 연구 방향 중 하나는 GPU를 지금보다 더 다양한 대기 분석 operator들에 적용해보는 것과 GPU 사용 시 발생하는 overhead를 줄이는 것이다. 또한, 하나의 노드에서 GPU를 여러 개 사용하는 경우와, 여러 노드에서 GPU를 사용하는 경우 발생하는 문제점들을 해결하는 것도 필요하다.

참고 문헌

[1] Shimada, Susumu, et al. "Coastal wind measurements using a single scanning LiDAR." Remote Sensing 12.8 (2020): 1347.

[2] Xian, Jinhong, et al. "Urban air pollution monitoring using scanning Lidar." Environmental Pollution 258 (2020): 113696.

[3] Matějíček, Luboš, Pavel Engst, and Zbyněk Jaňour. "A GIS-based approach to spatio-temporal analysis of environmental pollution in urban areas: A case study of Prague's environment extended by LIDAR data." Ecological Modelling 199.3 (2006): 261-277.

[4] Klett, James D. "Stable analytical inversion solution for processing lidar returns." Applied optics 20.2 (1981): 211-220.

[5] Pappalardo, Gelsomina, et al. "EARLINET: towards an advanced sustainable European aerosol lidar network." Atmospheric Measurement Techniques 7.8 (2014): 2389-2409.

[6] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: Simplified data processing on large clusters." (2004).

[7] Brown, Paul G. "Overview of SciDB: large scale array storage, processing and analysis." Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. 2010.

[8] Liu, Feng, et al. "GPU accelerated array queries: The good, the bad, and the promising." HP Laboratories, Palo Alto, CA, USA, Tech. Rep. HPL-2014-50 (2014).

[9] Marcin, Simon, and André Csillaghy. "Running scientific algorithms as array database operators: Bringing the processing power to the data." 2016 IEEE International Conference on Big Data (Big Data). IEEE, 2016.

[10] Marcin, Simon, and André Csillaghy. "Accelerating scientific algorithms in array databases with GPUs." 2017 IEEE International Conference on Big Data (Big Data). IEEE, 2017.

[11] 노영민, et al. "두파장 스캐닝 라이다 시스템을 이용한 고해상도 미세먼지 질량 농도 산출." 대한원격탐사학회지 36.6 (2020): 1681-1690.

[12] Cudre-Mauroux, Philippe, et al. "SS-DB: A Standard Science DBMS Benchmark." Extremely Large Databases Conference 2010.

[13] Taft, Rebecca, et al. "Genbase: A complex analytics genomics benchmark." Proceedings of the 2014 ACM SIGMOD international conference on Management of data. 2014.
[14] Oloso, Amidu, et al. "Implementing connected component labeling as a user defined operator for SciDB." 2016 IEEE International Conference on Big Data (Big Data). IEEE, 2016.
[15] Planthaber, Gary, Michael Stonebraker, and James Frew. "EarthDB: scalable analysis of MODIS data using SciDB." Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data. 2012.
[16] Battle, Leilani, Remco Chang, and Michael Stonebraker. "Dynamic prefetching of data tiles for interactive visualization." Proceedings of the 2016 International Conference on Management of Data. 2016.
[17] Baumann, Peter, et al. "The multidimensional database system RasDaMan." Proceedings of the 1998 ACM SIGMOD international conference on Management of data. 1998.
[18] Papadopoulos, Stavros, et al. "The tiledb array data storage manager." Proceedings of the VLDB Endowment 10.4 (2016): 349-360.
[19] Zalipynis, Ramon Antonio Rodriges. "Chronosdb: Distributed, file based, geospatial array dbms." Proceedings of the VLDB Endowment 11.10 (2018): 1247-1261.
[20] Choi, Dalsu, Chang-Sup Park, and Yon Dohn Chung. "Progressive top-k subarray query processing in array databases." Proceedings of the VLDB Endowment 12.9 (2019): 989-1001.
[21] Jiang, Li, Hideyuki Kawashima, and Osamu Tatebe. "Incremental window aggregates over array database." 2014 IEEE International Conference on Big Data (Big Data). IEEE, 2014.
[22] Kyoseung Koo, Juhun Kim, and Bongki Moon. "MISE: An Array-Based Integrated System for Atmospheric Scanning LiDAR." Proceedings of the 33rd International Conference on Scientific and Statistical Database Management. 2021.

Abstract

Optimization for Atmospheric Scanning LiDAR Analysis using Array Database

Juhun Kim
Computer Science and Engineering
The Graduate School
Seoul National University

Atmospheric scanning LiDAR analysis is the process that obtains the fine-grained mass result of the range area by analyzing the LiDAR signal data. Researchers have been using RDBMS and GIS, MATLAB, or Python to implement the process. It is possible to analyze a small amount of historical data from just a few LiDAR h/w with those methods. However, with the large amount of historical data from lots of h/w, the performance does matter. It is clear that researchers usually check the analysis results by changing the parameters. Therefore, it is needed to use an appropriate database by considering the data characteristics and analysis process to manage and analyze the data fast and easily.

In this paper, we apply an array database to solve the problem. Array database fits on atmospheric scanning LiDAR data because it has multidimensional features, and there are operators that can use locality between adjacent cells. Also, SciDB has shared nothing architecture for scalability. We implemented the analysis process with User-Defined Operators (UDOs) in SciDB and optimize them in two ways. First, we use parallel chunk processing in SciDB. When the operator is running, chunks inside the instances are processed in

parallel. Second, we implemented a GPU version of the operator which has many repetitive processes in parallel.

The experiments are held with the data based on real observed datasets. We show that our approach with the array database is better than the previous methods. Researchers can check the analysis results of big data in a short time with our method.

Keywords : SciDB, UDO, Parallel Chunk Processing, GPU
Student Number : 2019-26528

Acknowledgements

논문을 작성하기까지 도움을 주신 많은 분들께 감사의 말씀을 전합니다. 먼저, 지도 교수님이신 문봉기 교수님께 감사의 말씀을 드립니다. 처음 논문 주제를 고민하던 때부터 논문 방향을 정하고 추진해가는 동안에도 계속해서 조언을 해주시고, 격려해 주셔서 논문을 쓸 수 있었습니다. 많이 부족함에도 좋은 점들을 언급해주시고 긍정적으로 말씀해 주셔서 큰 힘이 되었습니다.

다음은 본 논문이 기반을 두고 있는 demo paper를 함께 작성한 교승씨에게 감사의 말씀을 드립니다. 본 논문은 해당 demo paper에 부가적인 아이디어를 추가하여 작성하였습니다. 본 논문의 내용 중 일부 아이디어와 결과들이 해당 논문에서 가져왔거나 바탕을 두고 있습니다. 해당 논문에서 작업했던 내용들이 본 논문을 작성하는데 큰 도움이 되었습니다.

또한, 본 논문의 아이디어를 생각할 수 있었던 배경에는 국토교통과학기술진흥원의 지원을 받아 진행된 “라이다 스캐닝을 이용한 미세먼지 모니터링 시스템 구축 및 실증” 과제가 있었습니다. 해당 과제에 참여하는 동안에 도움을 주신 많은 분들께 감사의 말씀을 드립니다.

마지막으로 DBS 연구실 사람들에게 감사의 말씀을 드립니다. 함께 연구실 과제를 수행하거나 수업을 들으면서 고민하고 소통할 수 있는 기회가 있어서 감사한 시간이었습니다. 많이 부족한 학생이었지만, 그럼에도 함께 해주셔서 감사했습니다. 연구실 안에서의 경험을 통해 여러 방면으로 많이 배울 수 있었습니다. 앞으로의 진로에서도 해당 경험을 바탕으로 더욱 성장할 수 있도록 노력하겠습니다. 감사합니다.