Journal of The Korea Society of Computer and Information
Vol. 18, No. 1, January 2013
www.ksci.re.kr
http://dx.doi.org/10.9708/jksci.2013.18.1.001

실시간 3차원 레이저 레이더 영상 생성을 위한 CUDA 기반 병렬처리 소프트웨어 설계

조 용 일*, 하 중 림*, 양 지 현*, 김 재 협*

The Design of Parallel Processing S/W Using CUDA for Realtime 3D Laser Ladar Imaging System

Yong Il Cho*, Choong Lim Ha*, Ji Hyeon Yang*, Jae hyup Kim*

요 약

본 논문은 3차원 레이저 레이더(LADAR, Laser Ladar) 영상 생성 시스템 개발을 수행함에 있어, 요구되는 실시간 처리를 구현하기 위해 CPU(Central Processing Unit) 및 GPU(Graphic Processing Unit)의 병렬처리 구조를 설계하는 CUDA(Common Unified Device Architecture) 기반 소프트웨어(SW, Software) 구현 기법에 대하여 설명한다. LADAR 시스템은 레이저 거리정보를 기반으로 3차원 영상을 생성하는 복잡도 높은 시스템으로써, 각 단계별로 많은 량의 처리 자원이 필요하다. 따라서, 한정된 시스템 자원 내에서 이를 실시간으로 처리하기 위해서는 반드시 병렬처리 구조를 설계 및 적용해야 한다. 본 논문에서는, 처리 알고리즘의 단계적 분석을 통해 분할 가능한 작업에 대하여 CUDA GPU로 할당 및 처리를 수행함으로써, 시스템에서 요구하는 실시간 처리를 달성하였으며, 처리 속도 분석을 통해 최대 46%의 처리 속도 향상을 확인할 수 있었다.

▸ Keywords : CUDA, 병렬처리, 3D 레이저 레이더

Abstract

In this paper, we propose a CUDA(Common Unified Device Architecture) based SW(software) design method for CPU(Central Processing Unit) and GPU(Graphic Processing Unit) parallel structure to implement real-time process in 3D Laser ladar(LADAR) imaging system. LADAR is a complex system to generate 3-dimensional image based on the laser ranging information, and requires massive process resources in each phase. Therefore, designing and implementing parallel structure are crucial to realize a real-time process within limited system resource. As a conclusion,

• 제1저자 : 조용일 • 교신저자 : 김재협
• 투고일 : 2012. 11. 20, 심사일 : 2012. 12. 03, 게재확정일 : 2012. 12. 08.
* 삼성탈레스(Samsung Thales)
www.kci.go.kr

we can meet the speed of required real-time process allocating separable work load to CUDA GPU by analyzing process algorithm in each phase and confirm the process speed increase by 46%.

▸ Keywords : CUDA, Parallel processing, 3D Laser Ladar

I. 서 론

레이저 기술의 발달과 IT 분야의 급속한 발전에 따라 다양한 민/군 분야에 레이저 기술이 사용되고 있다. 대표적으로, 군사용 레이저 거리 측정기(LRF: Laser Range Finder)는 이미 보편화되었으며, 전차에서 개인용 화기에 이르기까지 필수적인 전자장비로 자리 잡았다.

레이저 레이더(이하 LADAR, Laser Radar)는 레이저 거리 측정기와 동일한 원리를 이용하여, 3차원 영상을 얻는 장치이다. 레이저를 사용하여 일반 레이더와 비교하여 매우 높은 해상도를 얻을 수 있으므로, 비교적 미세한 대상체까지 식별이 가능하다. LADAR는 레이저가 물체로부터 반사되어 돌아오는 시간을 측정하여 3 차원 영상을 얻어 내는 센서이며, 매우 짧은 레이저 펄스를 사용하므로, 거리 방향의 높은 분해능과 거리에 따른 식별 능력이 매우 뛰어나다. 해외 선진국에서는 헬리콥터 충돌 방지용 센서로 우수한 성능이 입증되었고, 무인정찰기(UAV, Unmanned Aerial Vehicle) 에 장착되어 우수한 감시 정찰 기능을 인정 받았다, 국내에서는 최근 수년간 레이저 레이더의 개발 필요성이 대두되어 다양한 연구 개발이 진행되고 있다[1].

본 논문은 LADAR 시스템 개발을 위한 프로젝트 수행에서 산출된 소프트웨어(SW, Software) 시스템에 대하여 설명한다. 개발 중인 LADAR 장비는 레이저 조사를 위한 레이저를 생성하는 레이저 모듈, 반사된 레이저를 검출하여 거리정보로 변환하는 검출기, 시스템에서 설정된 시야각(FOV, Field Of View) 영역을 스캐닝하는 스캐너모듈, 검출기와 스캐너로부터 획득된 데이터를 이용하여 3차원 영상을 생성하는 로직모듈로 구성된다. 로직모듈에서는 스캐너와 검출기로부터 입력되는 대량의 데이터에 대한 연산 작업을 수행한다. 이러한 연산 작업은 영상처리보드와 그래픽처리보드를 이용하여 처리된다. 본 논문에서는 표기의 간략을 위하여, 영상처리 보드의 프로세서를 CPU로 표기하며, 그래픽 처리보드의 프로세서를 GPU로 표기한다. GPU는 CPU에서 처리하기 힘든 대용량의 데이터 연산을 수행하기 위해 사용된다.

본 논문에서는 로직모듈에서 수행되는 3차원 영상 생성 알고리즘에 대하여 CPU와 GPU 각각의 구현기법별 수행성능을 측정하고, 이를 토대로 시스템 요구조건에 만족하는 최적의 병렬처리 SW 구조 설계 결과에 대하여 설명한다. 고속 알고리즘의 실시간 처리를 위하여, 본 논문에서는 CPU와 GPU 각각에 프로세스를 할당하였으며, GPU의 병렬처리 구조를 구현하기 위하여 CUDA(Common Unified Device Architecture) 프로그래밍 기법을 활용하였다. CUDA GPU를 이용한 병렬처리 구조 SW설계는 다양한 시스템 설계 분야에 응용되고 있으며, 최근 고속 처리 시스템의 요구가 증가되면서 중요성이 대두되고 있다. 국내 산/학 분야에서도 활발한 연구개발이 진행되고 있으며, 대표적으로 Kim 등[2]은 선형대수 기법 구현에서 발생하는 많은 처리 과정을 CUDA GPU로의 분배 처리 설계를 통해 고속 처리 SW 구조를 설계하였으며, Kim 등[3]은 가우시안 혼합 모델을 모델링하는 과정에서 발생하는 다수의 중복 처리 단계에서 CUDA GPU를 활용하여 실시간 처리를 구현하였다.

본 논문에서는 2장에서 CUDA를 이용한 병렬처리 SW 설계 방법에 대하여 설명하고, 3장에서 3차원 LADAR 영상 생성 알고리즘에 대하여 설명한다. 4장에서는 알고리즘의 수행절차에 따른 CUDA 기반 구현에 대해 설명하고, 5장에서는 알고리즘 처리 수행 속도에 대한 CPU 및 CPU+GPU 구조의 비교분석한다. 6장에서 결론을 설명한다. 본 논문은 현재 개발중인 LADAR 시스템의 구현 내용으로써, 보안상의 이유로 주요 설계 수치는 xx로 표기하였다.

II. CUDA 기반의 병렬처리 SW 설계

CUDA는 NVIDIA에서 제공하는 GPU을 이용하여 병렬처리를 수행할 수 있는 SW 개발을 지원하는 프로그래밍 아키텍쳐이다. 대용량의 데이터 처리를 요구하는 응용에서 필요로 하는 병렬처리 SW를 C/C++과 같은 고수준의 프로그래밍 언어의 확장을 통해 개발의 편의성을 높이기 위해 고안되었다. CUDA를 지원하는 GPU는 고도로 쓰레드(Thread)화 되어있는 스트리밍 멀티프로세서(SM, Streaming Multiprocessor)의 배열로 구성되어 있고, 각 SM은 다수의 스트리밍 프로세서(SP, Streaming Processor)로 구성된

www.kci.go.kr

다. 또한 host(CPU)와 데이터 통신을 위한 별도의 메모리를 가진다. GPU 메모리는 사용 목적에 따라 전역/상수/공유 메모리와 레지스터로 이루어 있다.

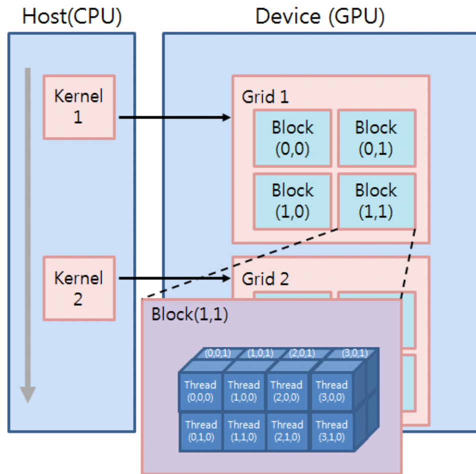


그림 1 CUDA Thread 구성
Fig. 1. Configuration of CUDA Threads

CUDA 프로그램은 한 개 이상의 영역으로 구성되며, 각 영역은 host(CPU) 혹은 device(GPU)에서 수행된다. 데이터 병렬성이 없는 영역은 host 코드로 구현되고 병렬성이 풍부한 영역은 device 코드로 구현된다. Device 코드는 ANSI C 코드에 커널(kernel)이라고 불리는 데이터 병렬 함수들과 관련된 자료구조를 명시하는 키워드를 확장한 형태로 작성된다. Kernel 함수가 개시되면 kernel은 디바이스로 옮겨져서 실행되며 데이터 병렬성을 처리하기 위해 많은 수의 thread를 생성한다. Kernel이 호출되어 수행될 때 생성되는 thread들을 모두 통칭하여 그리드(grid)라고 한다.

그림 1은 하나의 kernel이 수행될 때 GPU에서 생성되는 thread의 구성을 나타낸다[4][5]. 하나의 kernel이 수행되면 할당된 수 만큼의 thread가 생성되고 각각 동일한 작업을 병렬로 처리하게 된다. 이렇듯 병렬성이 존재하는 작업을 수많은 thread에 할당함으로써 처리성능을 극대화할 수 있다. 이러한 이점을 이용하여 현재 의학 화상(Medical Imaging), 유체동역학, 환경과학 등 많은 분야에서 사용되어 성능이 입증되었고, 점차 그 사용이 다른 많은 분야로 확장되어 가고 있다[6].

III. 3차원 LADAR 영상 생성 알고리즘

그림 2는 3차원 LADAR 영상 생성 알고리즘의 처리 흐름을 나타낸다. 3차원 LADAR 영상 생성 알고리즘은 데이터의 입력 및 정렬을 수행하는 데이터 정렬 모듈, 노이즈 제거를 위한 중복데이터 제거 모듈, 그리고 3차원 영상 생성 모듈과 같이 3 단계의 구조를 가진다. 데이터정렬 모듈은 3개의 서로 다른 입력 데이터의 시간적인 동기화를 통해 알고리즘 연산에 필요한 데이터를 생성하는 기능을 수행하며, 본 논문에서는 구현 기법에 대한 세부적인 기술은 언급하지 않는다.

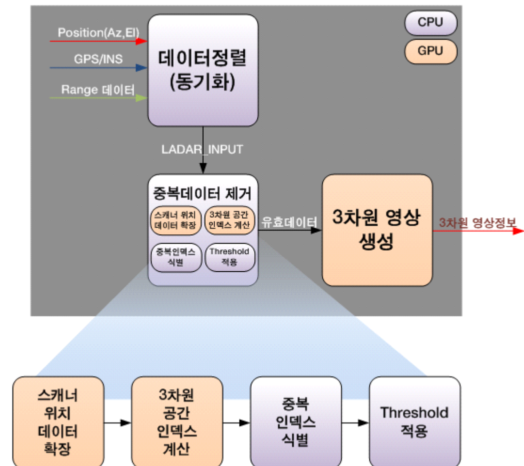


그림 2 알고리즘 처리 흐름도
Fig. 2. The flow chart of algorithm

중복데이터 제거 모듈은 크게 2가지 노이즈 성분을 제거하기 위해 사용된다. 첫 번째는 검출기에서 검출된 데이터 중 Dark Current, Solar Effect 등의 외부 노이즈 또는 검출기 특성에 의해 오검출되는 데이터이고, 두 번째는 스캔영역 중복에 의해 발생되는 중복데이터이다. LADAR 시스템에서 사용되는 노이즈 제거 알고리즘은 스캐너의 구동으로 인해 생성되는 수평/수직 FOV와 검출지역까지의 거리로 만들어지는 가상의 3차원공간에 대해 일정한 크기의 큐브를 생성하고, 하나의 큐브에 포함되어 있는 데이터의 수에 따라 유효데이터를 판단하는 기법을 사용한다. 이를 위하여, 중복데이터 제거 모듈은 스캐너 위치 데이터 확장, 인덱스 계산, 중복 인덱스 식별 그리고 문턱치(Threshold) 적용의 4가지 서브 모듈로 구성된다.

www.kci.go.kr

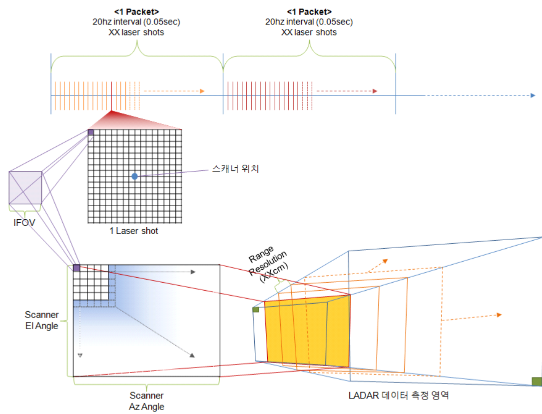


그림 3. LADAR 데이터 획득
Fig. 3. Data acquisition of LADAR system

스캐너 위치데이터 확장 모듈은 스캐너에서 수신된 위치 데이터를 검출기에서 수신된 데이터의 수만큼 확장하는 기능을 수행한다. 그림 3은 스캐너와 검출기에서 수신되는 데이터의 3차원적인 위치를 보여준다.

LADAR 시스템에서는 xx KHz의 속도로 레이저가 발사된다. 레이저 발사와 동시에 스캐너 또한 수평 xx.x Hz, 수직 x Hz의 속도로 스캔을 수행하며 레이저 모듈과 마찬가지로 xx KHz의 주기로 매 순간의 수평/수직 위치 데이터를 획득하게 된다. 이때, 위치 데이터는 검출기의 중심 위치만 획득된다. 따라서, 검출기의 중심 위치를 나타내는 스캐너 위치정보를 이용하여 검출기 모든 픽셀의 정확한 위치를 계산해야 한다.

인덱스 계산 모듈은 입력된 스캐너 위치와 검출기 거리정보를 이용하여 현재 데이터가 몇 번째 인덱스 번호를 가지는지 계산한다. 중복 인덱스 식별 모듈에서는 하나의 큐브 내에 존재하는 데이터의 수를 인덱스 번호를 이용하여 판단하고 저장한다. 이렇게 식별된 큐브별 데이터의 수를 시스템에 최적화된 threshold 값과 비교하여 노이즈 여부를 판단한다.

3차원 영상생성 모듈은 이렇게 얻어진 최종 데이터들에 대한 좌표변환을 수행한다. 스캐너의 수평/수직 위치정보를 바탕으로 구해진 픽셀의 위치 정보는 구좌표계이다. 이러한 각 픽셀의 좌표정보를 모니터 화면에 3차원 영상으로 전시하기 위해 직각좌표계로 변환한다. 그리고 최종적으로 LADAR 시스템의 자세변화에 따른 표적의 흔들림을 보정하기 위해 INS(Inertial Nevigation System) 데이터를 이용하여 회전변환을 수행한다.

IV. CUDA를 이용한 알고리즘 구현

1. 스캐너 위치데이터 확장

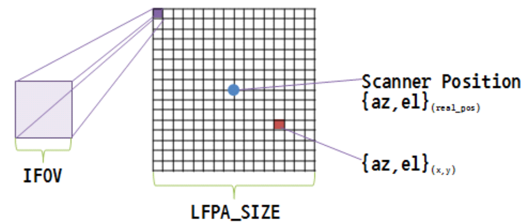


그림 4. 스캔 위치데이터 확장
Fig. 4. Extension of scan position

그림 4는 스캔 위치데이터 확장의 개념을 나타낸다. 실제 검출기의 크기는 n x n 크기이나, 실제 스캔되어 얻어지는 위치 정보는 정중앙의 정보만이 유일하다. 따라서, 스캔 위치데이터에 대비하여 검출기 각 픽셀에 해당하는 위치 정보를 계산해야 한다. 스캔 위치데이터를 확장하기 위해 먼저 검출기 픽셀 하나의 IFOV(Instantaneous Field Of View) 정보가 필요하다. LADAR 시스템의 광학계 설계에 따라 검출기 픽셀 하나가 차지하는 실제 크기로 정해지며, 이 설계 값을 이용하여 IFOV를 계산할 수 있다. 식 1은 IFOV를 계산하는 방법을 나타낸다.

$$IFOV = \frac{LFPA}{Target\ range} \tag{1}$$

LFPA는 laser focal plane array로써, 한 픽셀에 투영되는 실제 영역의 크기를 의미하며, target range는 광학계 검출기 픽셀과 실제 투영 영역간의 거리를 의미한다.

식 2는 식 1에서 계산된 IFOV를 이용하여 검출기 각 픽셀에 해당하는 수평/수직 각도 정보를 계산하는 방법을 나타낸다.

$$\{az, el\}_{(x,y)} = \{az, el\}_{realpos} + IFOV\left(position(x,y) - \frac{1}{2}LFPA\,SIZE\right) \tag{2}$$

$position(x,y)$는 계산하고자 하는 픽셀의 위치를 나타내며, LFPA SIZE는 검출기의 크기를 나타낸다.

www.kci.go.kr

2. 3차원 공간인덱스 계산

3차원 공간인덱스는 스캐너의 수평/수직 구동각도와 표적까지의 거리로 얻어지는 3차원 공간을 수평/수직 IFOV와 거리해상도를 기준으로 하는 여러 개의 작은 육면체들로 분할하여 각각의 육면체에 인덱스 번호를 부여하는 방법이다. 그림 5는 분할된 3차원 공간과 각 공간에 포함되는 데이터를 시각적으로 나타낸 것이다.

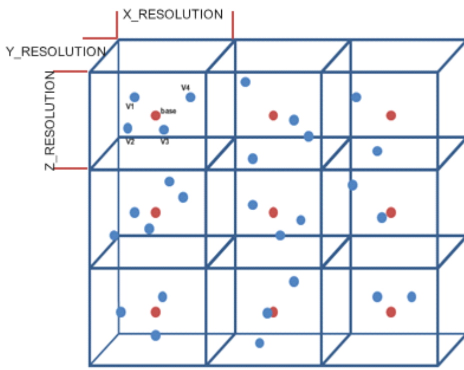


그림 5. 3차원 공간 분할
Fig. 5. 3D space decomposition

식 3은 각각의 측정 지점 픽셀을 3차원 분할 육면체에 할당하는 인덱스 I를 계산하는 방법을 나타낸다.

$$I = \frac{Target\ range}{Range\ resolution} \cdot NOC_x \cdot NOC_z + \frac{el_{(x,y)}}{IFOV} \cdot NOC_x + \frac{az_{(x,y)}}{IFOV} \quad (3)$$

Range resolution은 거리 해상도, 즉 검출기에서 식별 가능한 분할 거리 단위를 의미하며, NOC는 x, z 방향으로의 육면체의 개수를 의미한다. $el_{(x,y)}$ 및 $az_{(x,y)}$는 식 2에서 표현된 $\{az, el\}_{(x,y)}$의 각 요소를 의미한다.

위의 식을 전체 픽셀에 적용한 SW Pseudo code로 나타내면 아래 표 1과 같다.

표 1. 3차원 공간 분할 코드
Table 1. Pseudo code of 3D space decomposition

```
For k=0 to number_of_data

   For i = 0 to LFPA_SIZE
      el = elevation of real_pos
           + ( IFOV * LFPA_SIZE / 2 + i * IFOV )
      For j = 0 to LFPA_SIZE
        az = azimuth of real_pos
           + ( IFOV * LFPA_SIZE / 2 + j * IFOV )

        idx_el = el / IFOV
        idx_az = az / IFOV
        idx_range = Target range / Range resolution

        index = idx_range
              * ( NOC_X * NOC_Z )
              + ( idx_el * NOC_X ) + idx_az
```

위의 코드에서 볼 수 있듯이, 3개의 중첩구조를 이용하여 스캔 각도 정보와 각 데이터의 인덱스 번호가 계산된다. CPU는 한 번에 하나의 명령을 순차적으로 처리하기 때문에 위와 같은 중첩구조의 코드는 루프 횟수마다 반복하여 수행할 수밖에 없다. 즉, 하나의 코드 수행에 필요한 시간과 루프 횟수의 배수만큼의 시간이 필요하다. 물론 CPU 고유의 내부 instruction에 의해 여러 개의 연산이 동시에 처리가 가능하나 개념적으로 CPU는 순차적인 연산 작업과 외부 IO처리에 특화되어 설계되어 있어 병렬처리에 한계를 가진다.

반면에 GPU는 여러 개의 프로세서가 동일한 코드를 각각의 데이터를 이용하여 동시에 수행하기 때문에 이론적으로 GPU가 가지고 있는 프로세서의 수의 비례하는 연산성능을 가진다. 본 논문에서 설명하는 LADAR 시스템에서는 이를 위하여 CUDA를 기반으로 3차원 공간 분할 작업을 GPU로 할당하였다. 표 2는 CUDA GPU에 몇 개의 thread를 만들 것인지 알려주기 위한 변수선언문과 CUDA kernel 정의를 나타낸다.

www.kci.go.kr

표 2. CUDA Kernel 정의
Table 2. CUDA Kernel analysis

```
// 변수 선언문
dim3 dimGridIdx( 1, 625 );
dim3 dimBlockIdx( 512, 1 );

// CUDA Kernel 함수 정의
__global__
CalculateIndex<<<dimGridIdx, dimBlockIdx>>>
                ( float *in_position,
                  float *in_range,
                  float *out_position,
                  float *out_range,
                  float *out_index,
                  float *size_cubic )
```

위 변수선언문은 grid안에 1×625개의 블록(Block)이 존재하고, block 1개당 512×1개의 thread가 존재한다는 의미한다. 따라서, CalculateIndex() kernel에서 총 320,000개의 픽셀데이터를 처리한다는 의미가 된다. 데이터 처리를 몇 개의 block으로 나누어 처리할 것인지, block당 몇 개의 thread를 생성할 것인지는 GPU가 가지고 있는 자원을 고려하여 설계하였다. 무조건 많은 block과 thread를 동시에 수행하도록 설계를 한다면 GPU가 지원하는 범위를 넘어서서 오히려 수행성능에 더 안 좋은 영향을 미칠 수도 있기 때문이다.

3. 중복인덱스 식별 및 threshold

3차원 공간인덱스 연산을 통해 얻어진 각 픽셀데이터의 인덱스 번호를 통해 각 픽셀이 3차원 공간상에 어디에 위치하고 얼마나 많은 픽셀들이 한 공간 안에 모여 있는지 알 수 있다.

LADAR 시스템의 특성상 Solar Effect 또는 Dark Current에 의한 노이즈 발생 가능성이 있고, 스캐너 동작 방식으로 인해 여러 데이터들이 하나의 공간에 중복해서 존재할 확률이 높아진다. 이렇게 중복되는 픽셀들은 최종 결과인 3차원 영상에 악영향을 미치며, 연산시간을 증가시키는 원인이 된다.

이러한 노이즈 성분을 제거하기 위해 하나의 공간에 존재하는 픽셀의 수를 카운팅하고 threshold 값 이상의 픽셀데이터가 존재한다고 판단되는 공간 내의 픽셀데이터들만 최종 3차원 영상생성을 위한 데이터로 할당한다.

중복인덱스를 식별하는 방법은 매우 단순하다. 모든 픽셀데이터의 인덱스 값을 루프를 돌면서 동일한 인덱스 번호를 가지는 픽셀데이터의 수를 카운팅하면 된다. 아래 표 3은 중복인덱스를 검색하는 코드이다.

표 3. 중복 인덱스 검색 코드
Table 3. Code of replicated index retrieval

```
a. For k = 0 to number_of_data
b.     index_position[ index[ k ] ] =
              index_position[ index[ k ] ] + 1
```

이렇게 구해진 중복인덱스에 대해 threshold를 적용하는 코드는 아래 표 4와 같다.

표 4. Threshold 적용
Table 4. Application of threshold

```
a. final_pixel_count = 0

b. For k = 0 to number_of_data
c.    if index_position[ index[ k ] ] >= threshold
d.       final_pixel_scan_position[ final_pixel_count ]
                                = scan_position[ k ]
e.       final_pixel_range[ final_pixel_count ]
                                = range[ k ]
f.       final_pixel_count = final_pixel_count + 1
```

위의 코드에서 보듯이, 중복인덱스 검색과 threshold 적용 로직 또한 루프를 이용하여 픽셀의 수만큼 처리하는 방식이다. 그래서 두 기능 모두 CUDA GPU를 이용하여 처리가 가능할 것으로 판단될 수 있다. 그러나, 위의 두 기능에는 GPU로는 처리하는데 적합하지 않은 표 3의 b라인과 표 4의 f라인이 포함되어 있다.

위의 2개의 코드 모두 동일한 변수에 값을 변경하는 기능을 수행한다. 이러한 코드를 GPU에서 처리하게 되면, 여러 개의 thread가 동일한 구문을 동시에 접근하게 될 가능성이 높아진다. 이러한 경우 각각의 thread가 갱신되지 않은 변수값을 가지고 증가연산을 수행하여 최종단에서 잘못된 결과를 얻게 되는 원인이 된다. 이러한 문제를 방지하기 위해서는 하나의 thread가 해당 변수를 접근하는 중에 다른 thread는 자신이 사용할 수 있을 때까지 대기상태에 머무를 수 있도록 thread간의 동기화를 맞춰줘야 한다. 하지만 이러한 동기화 작업은 GPU를 통해 얻고자하는 이득을 상당부분 포기해야 하는 상황을 발생시킬 수 있으며, 순차적으로 연산을 처리하는 시간보다 더 많은 시간이 발생할 수도 있다. 이러한 이유로 LADAR 시스템에서는 중복인덱스 식별과 threshold 기능은 GPU에서 담당하지 않고 CPU가 직접 연산을 수행하도

www.kci.go.kr

록 할당하였다.

4. 3차원 영상 생성

노이즈가 제거된 최종 픽셀데이터는 좌표변환과 회전변환을 거쳐 3차원 영상으로 생성된다. 우선 위치데이터와 거리데이터로 이루어진 구좌표계 데이터를 직각좌표계로 변환한 후 INS 데이터를 이용하여 LADAR 장비의 자세보정을 위한 회전변환을 수행한다. 식 4는 구좌표계에서 직각좌표계로 변환하는 방법을 나타낸다.

$$\begin{aligned} x &= r * \cos(el) * \sin(az) \\ y &= r * \cos(el) * \cos(az) \\ z &= r * \sin(el) \end{aligned} \tag{4}$$

이렇게 변환된 좌표에 회전행렬을 적용한 최종 3차원 전시데이터는 아래 식 5와 같이 계산된다.

$$R_r = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(r) & -\sin(r) \\ 0 & \sin(r) & \cos(r) \end{bmatrix} \tag{5}$$

$$R_p = \begin{bmatrix} \cos(p) & 0 & \sin(p) \\ 0 & 1 & 0 \\ -\sin(p) & 0 & \cos(p) \end{bmatrix}$$

$$R_y = \begin{bmatrix} \cos(y) & -\sin(y) & 0 \\ \sin(y) & \cos(r) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R_y \cdot R_r \cdot R_p \cdot \begin{bmatrix} V_x \\ V_y \\ V_z \end{bmatrix}$$

3차원 영상생성은 입력되는 각각의 픽셀데이터 단위로 행렬연산만 수행되기 때문에 CUDA GPU에 할당하여 병렬 연산을 수행하도록 설계하였다.

V. 처리 속도 분석

아래 표 5와 표 6은 각 알고리즘 단계별로 CPU만으로 수행했을 때의 처리 속도와 CPU 및 GPU에 대한 작업 할당을 통해 수행했을 때의 처리 속도를 정리한 것이다. 스캐너 위치데이터 확장과 공간 인덱스 계산 모듈은 하나의 kernel로 구현/처리 되기 때문에 동시에 처리 속도가 산출되었으며, 표기의 간편화를 위하여 인덱스 계산 항목으로 표기하였다.

스캔위치데이터 1250개, 거리데이터 320,000개가 입력데이터로 사용되었다.

표 5. CPU 단독 처리 시간
Table. 5. Processing time on CPU only

단위 : μsec

인덱스 계산	중복인덱스 식별	threshold	3차원 영상 생성
6507	4721	3282	13216
6524	4732	4277	13190
5769	2167	2352	13267
6502	2572	2326	13193
6504	3631	2257	13210
6361.2	3564.8	2898.8	13215.2

표 6. CPU+GPU 처리 시간
Table. 6. Processing time on CPU+GPU

단위 : μsec

인덱스 계산			3차원 영상 생성		
CPU	GPU	C + G	CPU	GPU	C + G
839	811.42	1650.42	700	677.12	1377.12
833	804.77	1637.77	701	677.02	1378.02
833	804.22	1637.22	700	677.44	1377.44
836	806.75	1642.75	704	681.02	1385.02
834	805.98	1639.98	703	679.74	1382.74
835	806.63	1641.63	701.6	678.47	1380.07

위의 결과는 CUDA Command-Line Profiler를 이용하여 얻은 CPU와 GPU 각각의 수행시간이다. 중복인덱스 식별 모듈과 threshold 모듈은 병렬연산을 수행하지 않으므로 비교에서 제외하였다.

CUDA를 이용하여 연산을 수행할 경우, 위의 표에서도 나타나듯이 GPU 처리시간 외에 CPU 처리시간이 발생하게 되는데, 이는 CPU가 kernel함수를 호출할 때 발생하는 overhead 시간이다. 그러므로 kernel함수의 실제 수행시간(Walltime)은 CPU time + GPU time이 된다. 연산 횟수 별로 편차가 있기는 하지만, 평균값을 이용하여 비교해보면 인덱스 계산 모듈은 약 3.87 배, 3차원 영상 생성 모듈은 약 9.58 배 더 빠르게 처리됨을 확인할 수 있다. 이 결과를

www.kci.go.kr

전체 알고리즘 처리 시간으로 확장하면 아래 식 6과 같이 계산된다.

$$\begin{aligned} &CPU\ only \\ &= 6361.2 + 3564.8 + 2898.8 + 13215.2 \\ &\cong 26040\ \mu sec \end{aligned} \quad (6)$$

$$\begin{aligned} &CPU + GPU \\ &= 1641.6 + 3564.8 + 2898.8 + 1380.1 \\ &\cong 9485\ \ \mu sec \end{aligned}$$

단, 식 6에서 계산된 결과는 알고리즘의 단계별 처리 속도의 단순 비교를 통해 산출된 결과를 의미한다. LADAR 시스템 전체의 처리 속도를 비교하게 되면, 추가적으로 CPU와 GPU간의 통신 처리 시간을 고려해야 한다.

LADAR 시스템에서 사용하고 있는 NVIDIA GT240 GPU는 128bit memory interface를 지원하며 device to device memory 복사의 경우 최대 25.2GB/s의 속도를 지원한다. 하지만 이 경우는 GPU device 내부에서의 데이터 전송 속도이고, host(CPU)와의 통신 속도는 이에 크게 못 미친다. 아래 표는 GPU의 실제 데이터 통신 속도를 측정한 것이다.

표 7. 데이터 전송 속도
Table. 7. Data transmission speed

Size	Direction	Performance (MB/s)
32GB	Host to Device	2532.4
	Device to Host	2189.7
	Device to Device	21572.6

표 8. CPU 및 GPU간 데이터 전송량
Table. 8. Data transmission capacity between CPU and GPU

방 향	용 도
C → G	인덱스 계산 모듈 입력
	① 스캐너 위치데이터(수평/수직) • 8 * 1250 = 10000 ≒ 9.766 KB ② 검출기 거리정보 • 256 * 4 * 1250 = 1280000 ≒ 1.22 MB
C ← G	인덱스 계산 모듈 출력
	③ 확장된 스캐너 위치데이터(수평/수직) • 256 * 8 * 1250 = 2560000 ≒ 2.44 MB ④ 확장된 검출기 거리정보 • 256 * 4 * 1250 = 1280000 ≒ 1.22 MB ⑤ 공간 인덱스 정보 • 256 * 4 * 1250 = 1280000 ≒ 1.22 MB
C → G	3차원 영상 생성 모듈 입력
	⑥ 중복픽셀 제거된 스캐너 위치데이터 • 8 * 86919 = 695352 ≒ 679.05 KB ⑦ 중복픽셀 제거된 검출기 거리정보 • 4 * 86919 = 347676 ≒ 339.53 KB ⑧ INS 회전변환 행렬 • 4 * 9 = 36 Byte
C ← G	3차원 영상 생성 모듈 출력
	⑨ 최종 Vertex 데이터 • 12 * 86919 = 1043028 ≒ 0.99 MB

표 7에서는 host(CPU)와 device(GPU) 간의 가능한 데이터 통신 대역폭을 나타낸다. GPU 내부 메모리간의 최대 대역폭은 약 21GB/sec 에 이르며, 이는 대용량 영상 처리에서도 데이터 통신 속도가 전체 시스템 처리 속도에 영향을 주지 않는 수준임을 알 수 있다. 그러나, CPU-GPU간의 데이터 통신 대역폭은 약 2.5GB/sec에 불과하며, 대용량 영상 데이터를 처리할 경우 초당 수십장 내외의 고속 처리 시스템에서는 무시할 수 없는 수준이다. 따라서, CUDA GPU기반의 병렬처리 구조를 설계하고, 처리 성능을 분석할 경우 반드시 CPU-GPU간의 데이터 통신 속도를 고려해야 한다.

표 8은 CUDA GPU로 할당된 각각의 알고리즘 단계에서 측정된 CPU-GPU 간의 데이터 통신량을 나타낸다. 표에서 C와 G는 표기의 편의성을 위하여 CPU와 GPU를 의미한다. 전체 알고리즘의 수행에 대하여 CPU와 GPU 간에는 총 4단계에 걸쳐 9번의 메모리 복사 작업이 이루어지며, 총 전송 데이터량은 약 8.1MB 이다. CUDA Command-Line Profiler를 이용하여 측정한 결과, 4465μsec의 시간이 메모

www.kci.go.kr

리 복사를 위해 사용되었음을 알 수 있었다. 따라서, CPU+GPU SW구조에 따른 수행에서는 총 13950 μsec의 처리 시간이 소요되었으며, 이는 전체 알고리즘 수행시간의 약 32%가 메모리 복사에 사용된다는 것을 의미한다. 이를 토대로 분석했을 때, CPU 단독 처리 구조에 비해 약 46% 가량의 처리 속도 향상을 결론할 수 있다.

Ⅵ. 결론

본 논문에서 설명하는 LADAR 시스템의 로직모듈에서 수행되는 3차원 영상생성 알고리즘은 입력데이터를 처리하여 3차원 좌표정보 생성하고, 생성된 데이터를 전시장비로 전송하는 일련의 작업들을 주어진 시간 안에 안정적으로 수행한다. 실험 결과 CPU 단독처리 구조에 비해 약 46% 가량의 처리 속도 향상을 확인하였으며, 이는 GPU를 이용한 병렬처리 구조 도입으로 인해 연산시간을 감소시킨 결과이다. 그러나, 복잡도 높은 시스템 구조 설계에 있어 추가적인 기능 구현과 보완을 위해서는 단계별 처리 과정에 대한 최적화 작업에 대한 연구 개발이 지속적으로 이루어져야 한다.

이와 관련하여 실험 검토를 통해 추가적인 최적화 방안에 대해서 설명한다. 가장 먼저 생각해 볼 수 있는 부분은 중복 인덱스 식별 기능과 threshold를 이용한 노이즈제거 기능을 들 수 있다. 4.3절에서 언급했던 것처럼 이 두 기능은 연산에 이용되는 모든 데이터들이 하나의 변수에 접근을 시도해야 한다는 이유로 CUDA를 이용한 병렬처리에서 제외되었다. CPU에 의해 처리되는 이 두 기능의 연산시간은 전체 알고리즘 수행 시간의 68%에 달한다. 이 기능들을 병렬로 처리할 수 있다면 처리성능을 대폭 향상시킬 수 있을 것으로 판단된다. 여러 개의 thread가 하나의 메모리에 접근하려고 할 때, 그 메모리에 대한 원자적(Atomic) 접근이 보장되어야 신뢰할 수 있는 최종 결과를 얻을 수 있다. CUDA에서는 이러한 원자적 접근을 보장하기 위한 API 함수들을 지원한다. 하지만 단순히 이러한 API 함수들을 해당 구문과 1:1로 맞교환해서는 수행된 결과는 올바른 값을 얻을 수 있겠지만 성능향상을 기대할 수 없다. 접근을 시도하는 수많은 thread에 대해 GPU에서 추가적인 스케줄링 시간이 추가되기 때문이다. 이러한 성능저하를 막기 위해서는 동시에 메모리에 접근하는 thread의 수를 줄여 줌으로써 부가적인 비용발생을 억제할 필요가 있다[7]. 또한 동시 접근하는 thread의 수를 줄이기 위해 작은 데이터 단위로 수행이 가능하도록 알고리즘을 세분화할 필요가 있다. 이러한 원자적 접근이 가능한 알고리즘의 설계가 이루어진다면 매우 큰 성능향상을 기대해 볼 수 있다.

다음으로 고려해야 할 사항은 데이터 전송시간이다. 현재 LADAR 시스템에서 CPU와 GPU간 데이터 전송에 소모되는 시간은 약 4500μsec이다. 전체 알고리즘 수행시간을 감안했을 때, 무시할 수 없는 시간이라는 것은 확실하다. 또한 현재 전송되는 데이터의 양이 약 8MB 정도이다. 하지만 LADAR 구성품의 성능개선, 특히 검출기의 성능개선을 통해 한 번에 획득되는 데이터의 수가 증가할 경우, 로직모듈에서 처리해야 하는 데이터의 양은 지금보다 훨씬 많이 증가할 수 있다. 이렇게 데이터의 양이 증가하게 되면 데이터 전송에 따르는 비용 또한 증가하기 된다. 메모리 복사에 작업에 사용되는 cudaMemcpy() 함수는 CPU에 의해 호출되면 복사작업이 완료되기 전까지는 다시 CPU로 반환되지 않는다. CPU는 복사작업이 완료될 때까지 대기상태에 들어가게 된다. CPU가 다른 작업을 처리하지 못하는 시간이 발생하게 되는 것이다. 복사되는 메모리의 크기가 크면 클수록 지연시간은 늘어가게 된다. 이러한 CPU와 GPU간의 데이터 전송시간에 따른 지연시간을 줄이는 방법으로, cudaMemcpyAsync() 비동기 함수를 이용할 수 있다. 이 함수를 호출하면 제어권은 바로 CPU로 반환되고 GPU 내부적으로 메모리 복사작업이 이루어진다. CPU는 반환된 제어권을 받아 다른 작업을 수행할 수 있다. CPU와 GPU의 병렬 작업이 종료되고 다시 둘 간의 동기를 맞추기 위해 cudaThreadSynchronize() 함수를 이용해 GPU의 모든 작업이 끝날때까지 대기한다. 이와 같은 비동기식 함수를 사용하기 위해서는 메모리 복사를 위해 'pinned'된 메모리 영역이 사용되어야 한다. DMA(Direct Memory Access)를 통해 고속으로 데이터를 통신하기 위해 고정되고 정렬된 물리적인 메모리 영역이 필요하기 때문이다. 이러한 고정된 메모리 영역을 사용하면 데이터 전송 대역폭을 크게 향상시킬 수 있다. 이러한 비동기식 메모리 복사 API를 이용하면 CUDA stream 기능 또한 사용할 수 있다. CUDA stream이란 GPU에서 실행되는 함수들을 순차적으로 쌓아 놓은 큐로 정의할 수 있다. 현재 LADAR 시스템에서 수행된 CUDA 코드는 하나의 Stream을 이용하여 동작한 것으로 생각할 수 있다. Stream을 기능에 맞게 여러 개를 생성하고 각각의 stream이 동시에 실행될 수 있도록 kernel함수 및 메모리 복사 API를 배치할 수 있도록 SW를 설계한다면 CPU나 GPU의 유휴시간을 최소화함으로써 시스템의 성능을 향상시킬 수 있다.

이러한 다양한 측면에 대하여 고려된 SW 구조를 설계하기 위해서는 GPU 하드웨어 자원의 제약, 알고리즘 수행 흐름에 따른 기능 적용 제약, SW 복잡도 증가 등 CUDA GPU를 사용하기 위해 고려해야 할 사항이 다양하게 발생한다. 그럼에

www.kci.go.kr

도 불구하고 GPU 프로그래밍은 알고리즘 수행시간 단축이라는 커다란 이점을 가져올 수 있으며, 이는 실시간 시스템에서 시간적인 제약으로 가능하지 않았던 많은 부분들이 CUDA GPU를 적용함으로써 가능해 질 수 있음을 의미한다.

참고문헌

〔1〕 Jong Pil Ra, Jin Shin Ko, Min Sik Cho, Jang Jae Lee, and Eung Chul Kang, "3D Imaging Laser Radar(LADAR) System for Mapping using a High Repetition Rate Fiber Laser," Proc. of 19th Conf. on COOC, pp. 316-317, May 2012.

〔2〕 Kim Jung Hwan and Kim Jin Soo, "Implementation of Efficient Power Method on CUDA GPU," Journal of KSCI, Vol. 16, No. 2, pp. 9-16, Feb. 2011.

〔3〕 In-su Kim and Hyung-Il Choi, "Object Tracking Based on Gaussian Mixture Model Algorithm by Using Cuda," Proc. of KSCI, Vol. 19, No. 1, pp. 273-275, Jan. 2011.

〔4〕 http://www.nvidia.co.kr/object/cuda-kr.html, NVIDIA CUDA Programming Guide.

〔5〕 David B. Kirk and Wen-Mei Hwu, "Programming Massively Parallel Processors," Morgan Kaufmann Publishers, 2010.

〔6〕 Ho Young Lee, JongHyun Park, and JunSeong Kim, "Parallel Computation of FDTD algorithm using CUDA," Journal of IEEK CI, Vol. 47, No. 4, pp. 82-87, July 2010.

〔7〕 Jason Sanders, Edward Kandrot, and Jack Dongarra, "CUDA by Example," Addison-Wesley Professional, 2010.

저 자 소 개

조 용 일

1998: 삼성전자
전략시스템팀 연구원

현 재: 삼성탈레스
광전자연구소 선임연구원

관심분야: 컴퓨터공학, SW설계

Email : yongil007.cho@samsung.com

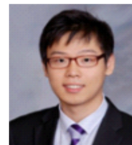
하 중 림

2003: 홍익대학교
전자전기공학과 공학사.

현 재: 삼성탈레스
전문연구원

관심분야: 컴퓨터공학, HW설계,
레이저시스템

Email : choonglim.ha@samsung.com



양 지 현

1989: 성균관대학교
컴퓨터교육학과 이학사.

현 재: 삼성탈레스 연구원

관심분야: 컴퓨터공학, SW설계

Email : jihyeon.yang@samsung.com

김 재 협

2001: 한양대학교
전자계산학과 공학사.

2003: 한양대학교
컴퓨터공학과 공학석사.

2008: 한양대학교
컴퓨터공학과 공학박사

2008: 한양대학교
엠비언트인텔리전트SW연구팀
박사후연구원

현 재: 삼성탈레스 전문연구원

관심분야: 컴퓨터비전, 패턴인식

Email : jaehyup.kim@samsung.com

www.kci.go.kr