Journal of Korea Multimedia Society Vol. 19, No. 8, August 2016(pp. 1297-1309)
http://dx.doi.org/10.9717/kmms.2016.19.8.1297

대규모 레이더 신호 데이터의 실시간 분석을 위한 GPU 기반 객체 추출 기법

강 영 민†

GPU-based Object Extraction for Real-time Analysis of Large-scale Radar Signal

Young-Min Kang†

ABSTRACT

In this paper, an efficient connected component labeling (CCL) method was proposed. The proposed method is based on GPU parallelism. The CCL is very important in various applications where images are analysed. However, the label of each pixel is dependent on the connectivity of adjacent pixels so that it is not very easy to be parallelized. In this paper, a GPU-based parallel CCL techniques were proposed and applied to the analysis of radar signal. Since the radar signals contains complex and large data, the efficiency of the algorithm is crucial when realtime analysis is required. The experimental results show the proposed method is efficient enough to be successfully applied to this application.

Key words: GPU-parallelism, Connected Component Labeling, Radar Signal

1. 서 론

본 논문은 매 주기마다 16 기가 바이트 규모의 대규모 이미지가 생성되는 레이더 신호를 효율적으로 처리하고, 이 데이터에서 실시간으로 객체를 추출하여 다양한 처리를 할 수 있게 하는 고속 이미지 처리 기법을 다룬다. 이러한 목표를 달성하기 위하여 GPU의 병렬처리 속성을 활용하며, 연결된 객체를 효과적으로 추출하기 위하여 연결 요소 레이블링 (CCL, connected component labeling) 기법을 GPU를 이용하여 구현한다.

연결 요소 레이블링 기법은 하나의 이미지 내에서 인접한 픽셀 관계로 연결되어 있는 픽셀 그룹마다 별도의 레이블을 부여하는 기술이다. 이 기술은 컴퓨터 비전에서 핵심적인 기초 작업으로 요구된다. GPU가 다양한 영상처리에 사용되고 있지만[10], GPU를 활용한 효율적인 병렬처리 기술이 이 분야에 아직 제대로 적용되지 못 하고 있다[1].

GPU를 이용한 CCL 기법 가운데 이미지를 작은 타일로 구분하여 처리하는 방식이 제안되었고[2], 행 단위로 스레드를 할당하여 계산하는 방법도 제안되었다[3]. 또한 레이블의 동일성을 레이블 집합 단위로 다루는 방식의 기법 등도 제안되었다[4,7]. 그러나 이러한 기법들은 여전히 연결된 이웃들을 검사하는 방식으로 병렬성을 최대로 활용하지 못 하고 있다.

연결 요소 레이블링을 병렬 처리 방식으로 다루는 것은 쉬운 문제가 아니다. 연결 요소를 파악하는 것은 오래전에 알려진 바와 같이 병렬처리 방식만으로 구현될 수는 없다[5]. 그러나, 병렬 컴퓨팅 환경에

※ Corresponding Author : Young-Min Kang, Address: (48520) Sinseon-no 428, Nam-gu, Busan, Korea, TEL : +82-51-629-1253, FAX : +82-51-629-1249, E-mail : ym-kang@tu.ac.kr
Receipt date : Jul. 25, 2016, Approval date : Aug. 3, 2016

† Dept. of Game Engineering, Col. of Engineering, Tongmyong University
※ This research was supported by Tongmyong University Research Grant 2014 (2014A038).

서 이 문제를 해결하기 위한 노력이 계속 있어왔다[6]. 이러한 기법들은 병렬적 구조를 활용하여 계산상의 이익을 얻기 위하는 데에 집중했지만, GPU와 같이 대규모 데이터 병렬 구조에 적합한 방법을 제안하지는 못 하였다. 또한 연결 객체의 정보를 효율적으로 처리하기 위해 이러헌 연결 요소의 개수를 파악하고 바운딩 박스를 추출하는 등의 작업을 효율적으로 처리하는 문제는 일반적으로 연구의 대상이 아니었다.

이 논문에서는 대규모 데이터 병렬처리가 가능한 GPU 환경에 적합한 CCL 기법을 제안하고 각 단계별로 요구되는 스레드 커널 함수의 구현 방법을 설명한다. 아울러 연결 요소 각각의 바운딩 박스를 추출하는 등의 작업도 적절한 동기화 기법과 GPU의 병렬성을 활용하여 효율적으로 수행할 수 있는 방법을 제안한다. 이렇게 구현된 기법은 4096×4096 크기의 대규모 레이더 이미지에 적용하였다. 이전의 연구들에서는 이러한 대규모 영상에 대한 CCL 작업을 일반적으로 고려하지 않았다.

2. 원천 데이터의 구조와 실시간 가시화

2.1 원천 데이터

레이더 신호는 Fig. 1의 (a)와 같은 형태로 저장되어 있다. 총 4096개의 방위각(azimuth)의 신호가 저장되어 있으며, 하나의 방위각 데이터는 각각 4096개의 강도(intensity) 신호로 구성된다. 따라서 원천 데이터는 4096×4096 규모의 이미지로 볼 수 있다. Fig. 1의 (b)는 일부 영역의 신호를 확대한 것이다. 신호는 실제 객체뿐만 아니라 다양한 잡음(noise)가 함께 존재한다.

2.2 실시간 가시화

입력되는 레이더 신호의 규모는 매 프레임마다 16 기가바이트가 된다. 따라서 이러한 데이터를 빠르게 화면에 재생하는 기술이 필요하다. 이것은 입력신호를 오픈지엘(OpenGL)의 정점 버퍼로 보내어 바로 출력이 되도록 한다.

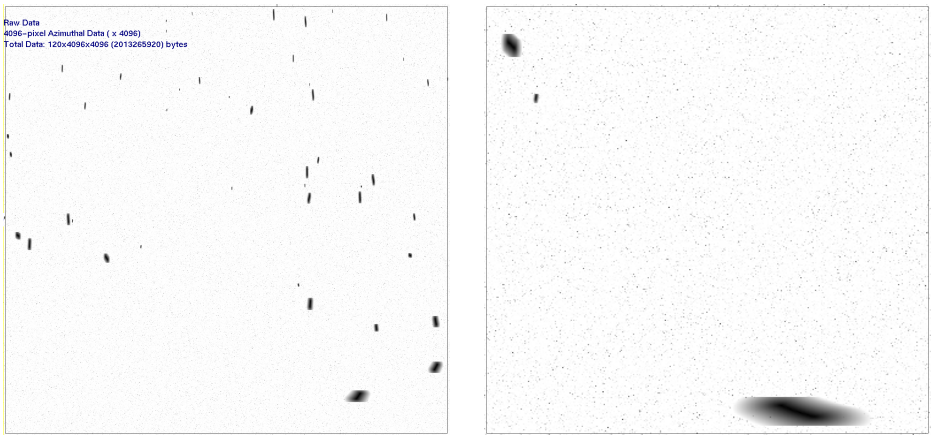
원천데이터를 가시화하는 것은 간단한 일이다. 오픈지엘(OpenGL) 정점 버퍼를 준비하고 총 4096×4096개의 정점정보를 담을 수 있도록 한다. 원천 데이터 I_σ는 4096 개로 구분된 방위각 각각의 데이터가 4096 개의 픽셀 정보로 들어온다. 이를 실세계 좌표계로 변화하여 4096×4096 정점 버퍼에 보낸 뒤에 가시화한다. 우리가 처리해야 하는 이미지 공간은 이렇게 가시화를 위해 변환된 이미지이며, 이 이미지를 I_χ라 하자.

I_σ와 I_χ가 모두 $w \times h$ 화소를 갖는 정방형 이미지라고 가정하자. I_χ의 특정 픽셀 $I_\chi^{x,y}$의 광강도는 다음과 같이 결정된다.

$$I_\chi(x,y) = I_\sigma\left(w\sqrt{x^2+y^2},\ h\frac{\cos^{-1}(x/r)}{2\pi}\right) \quad (1)$$

2.3 잡음 처리

입력된 레이더 정보는 다양한 잡음을 포함하고



(a) (b)

Fig. 1. Input source data. (a) whole data (b) Enlarged image of partial data.

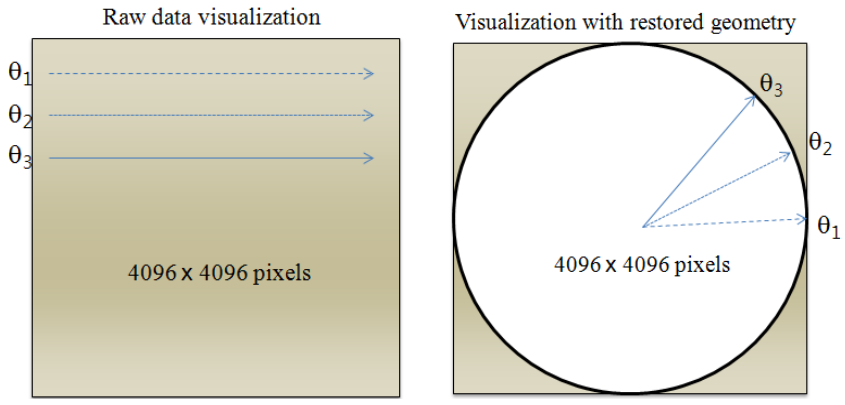


Fig. 2. Visualization of source data and transformation to the real-world coordinate. (a) source data image I_σ (b) Image I_χ transformed to the real-world coordinate.

있다. 이러한 잡음을 제거하는 일은 GPU를 이용하여 쉽게 할 수 있다. GPU 쓰레드는 I_χ의 픽셀 수만큼 생성되며, 각각 노이즈를 제거하기 위한 컨벌루션 필터를 적용한다. 이렇게 잡음을 제거하여 얻은 이미지를 I_ϕ라 하자. 이 이미지의 각 픽셀은 완전히 독립적으로 계산될 수 있으므로 병렬화가 매우 용이하며 빠르게 얻을 수 있다.

Fig. 3은 이러한 잡음 제거의 결과를 보이고 있다. 그림의 (a)는 수신된 레이더 신호를 그대로 가시화한 I_χ이며, (b)는 잡음이 제거된 I_ϕ이다. 이때 각각의 픽셀에는 컨벌루션 필터가 적용되며, 이 컨벌루션 필터를 통해 얻어진 데이터가 지정된 임계치를 넘을 경우 1 그렇지 않을 경우 0으로 처리하여 잡음을 제거하였다.

3. 연결요소 추출법

잡음을 제거하는 작업은 각각의 픽셀 단위로 스레드를 생성하여 처리하면 빠르게 병렬처리가 가능하다. 컨벌루션 필터를 적용하는 작업이 상호독립적이기 때문이다. 그런데 이렇게 얻은 이미지에서 연결요소를 추출하는 일은 병렬화가 어렵다. 가장 단순한 형태의 연결 요소 추출 방법은 값을 가진 픽셀 하나에서 인접 관계(adjacency)로 연결된 모든 픽셀을 그래프 탐색 방식으로 찾는 것이다. 그러나 이러한 방식은 병렬 알고리즘으로 구현할 수가 없다. 본 논문에서는 이러한 문제를 해결하고 GPU의 병렬성

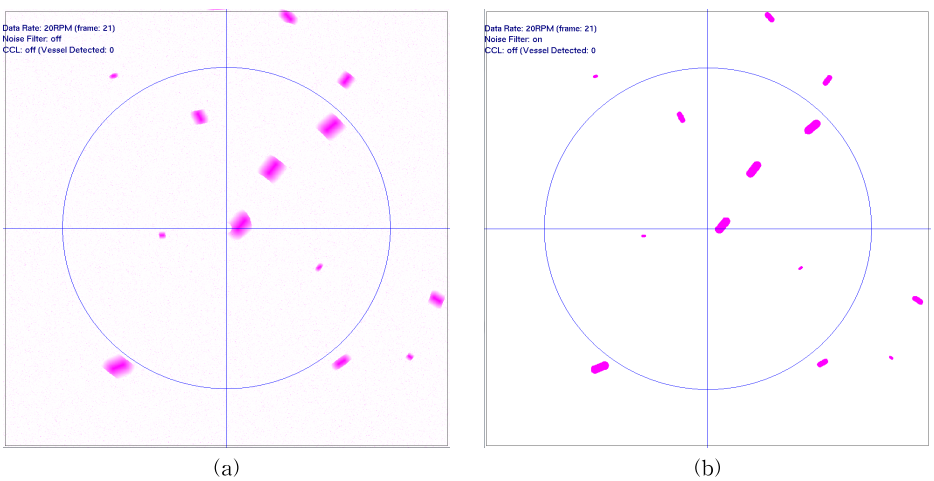


Fig. 3. Denoised image I_ϕ obtained from the radar image I_χ. (a) visualized radar image I_χ (b) denoised image I_ϕ.

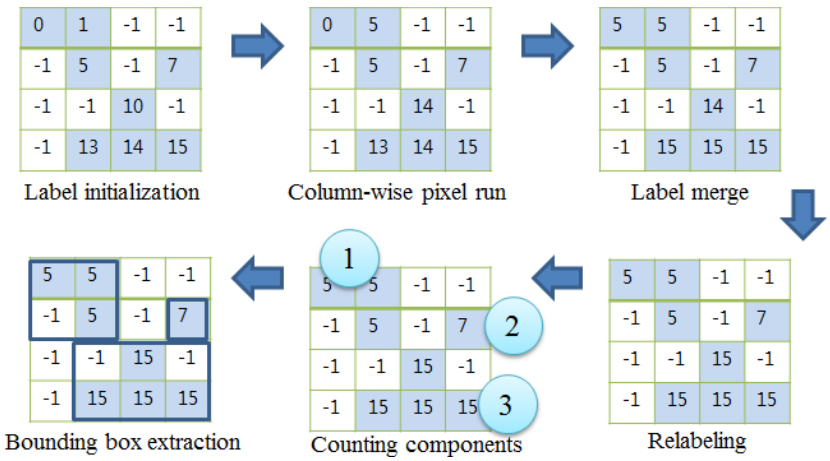


Fig. 4. Image processing phases for the proposed GPU-based connected component labeling.

을 활용하여 연결요소를 빠르게 추출할 수 있는 기법을 제안한다.

제안된 기법은 다음과 같은 단계로 이루어진다.

- 초기 레이블 부여
- 열(column) 단위 레이블 수정
- 레이블 병합
- 레이블 재지정
- 연결요소 계수
- 연결요소 바운딩 박스 계산

각 단계에서 수행하는 작업의 결과를 Fig. 5에서 확인할 수 있다. 레이블 초기화는 켜진 픽셀과 켜지지 않은 픽셀로 구분하고, 켜진 픽셀은 이미지 내의 위치를 자신의 레이블로 설정한다. 그림의 예는 4×4 이미지이므로 16개의 픽셀을 가지고 0에서 15까지의 레이블이 각각의 픽셀에 할당된다. 켜지지 않은 픽셀은 레이블이 -1이다.

연결요소 추출의 가장 기본적인 개념은 연결된 픽셀들이 동일한 레이블을 갖도록 하는 것이다. 그

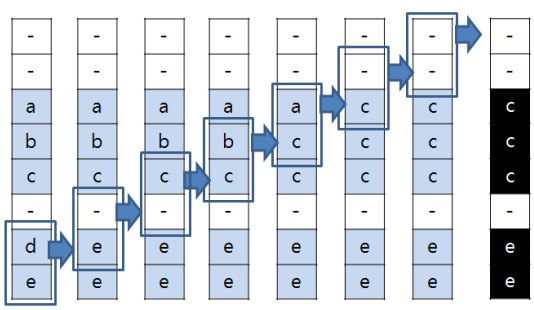


Fig. 5. Column-wise adjacent pixel run labeling.

러나 연결된 픽셀들을 그래프 노드의 인접성을 따라 순회하는 방식으로 구현할 경우 병렬 처리가 불가능하다. 이러한 문제를 해결하기 위해 열 단위로 연결된 픽셀들의 레이블을 통합하고, 이미지 전체의 레이블을 통합한 뒤 최종적으로 연결요소 별 레이블을 얻을 수 있다. 분석을 위해서는 연결요소의 개수를 세는 작업과 바운딩 박스를 추출하는 작업도 함께 이루어진다.

3.1 초기 레이블 부여

레이더 데이터를 구성하는 픽셀은 반사된 전파에 의해 감지된 픽셀과 그렇지 않은 픽셀로 구분할 수 있다. 연결된 픽셀들을 하나의 객체로 인식하기 위해서는 연결된 픽셀들이 하나의 레이블을 갖도록 한다. 이러한 목표를 달성하기 위해 각각의 픽셀에 초기 레이블을 설정하는데, 이 초기 레이블을 설정하는 방법은 각각의 픽셀의 위치에 근거하여 좌상단 첫 화소의 레이블을 0으로 하고 우하단 마지막 화소의 레이블은 이미지를 구성하는 화소의 수가 n일 때, $n-1$로 설정한다.

이러한 작업은 GPU 스레드를 생성하여 간단히 병렬적으로 처리할 수 있다. 알고리즘 1은 너비가 w 픽셀, 높이가 h 픽셀인 이미지에 대해 초기 레이블을 지정하기 위해 $w \times h$ 개 생성되는 스레드가 수행하는 작업이다. 이때 I_λ^i는 i 번째 화소로 $i\%w$ 행, i/w 열 위치의 화소를 의미한다. 즉 $I_\lambda^i = I_\lambda^{(i\%w, i/w)}$를 의미한다.

Algorithm 1. Thread kernel function for label initialization for each pixel

kernel **initLabel** (input: $I_\phi \in \mathbb{Z}^{w\times h}$, output: $I_\lambda \in \mathbb{Z}^{w\times h}$) :
$threadID$: $[0,\ w\times h-1]$
if($I_\phi^{threadID} > \theta$) then $I_\lambda^{threadID} = threadID$
else $I_\lambda^{threadID}$ = 'off'

이 작업은 하나의 픽셀 레이블을 지정하는 데에 다른 픽셀의 레이블을 고려할 필요가 없기 때문에 병렬로 처리하는 데에 어떠한 문제도 없다.

3.2 열 단위 레이블 수정

앞서 언급한 바와 같이 연결 요소를 추출하기 위해서는 인접성(adjacency) 관계로 연결된 화소 집합이 하나의 레이블을 갖도록 변경해야 한다. 이러한 작업을 그래프(graph) 기반의 연결 노드 순회 방식으로 수행할 경우 병렬 처리가 불가능하다. 이러한 문제를 해결하기 위해 우리는 이미지 공간에서 인접한 화소들의 레이블을 적절히 갱신하는 작업을 병렬적으로 수행하여 하나의 연결요소가 하나의 레이블을 갖도록 한다.

이러한 작업의 가장 기초가 되는 작업은 하나의 행(row)이나 열(column) 내에서 이어진 화소들을 판단하는 것이다. 이렇게 한 줄로 연결된 화소의 연속을 찾는 작업이 필요하다. 이를 위해 이미지 I_λ가 가진 너비 w 만큼의 스레드를 생성하고, 각각의 스레드는 해당 열에서 이 연속 화소들이 하나의 레이블을 갖도록 갱신한다. 이때 연속 화소들이 갖는 레이블은 이들 가운데 가장 큰 값을 가진 레이블로 설정한다. 이 작업은 하나의 열에서 가장 아래쪽 화소 바로 위의 화소에서 시작하여 위로 하나씩 스캔(scan)하며 해당 화소가 바로 아래 화소와 연속되는 경우에 아래 화소의 레이블로 갱신하면 된다. 하나의 스레드가 이미지의 높이 h 만큼의 비교를 순차적으로 수행해야 하지만 레이더 데이터 자체가 매우 큰 w를 갖기 때문에 병렬성이 저하되어 속도가 떨어지는 문제는 크게 발생하지 않는다.

Fig. 5는 알고리즘 2의 열 단위 연속화소 추출이 하는 일을 시각적으로 보이고 있다. 각각의 스레드가 동작을 마치면 Fig. 5의 가장 오른편에 나타난 바와 같이 각 스레드가 맡고 있는 열 안에서 연속하여 나타나는 화소 그룹별로 동일한 레이블을 갖게 된다.

Algorithm 2. Thread assigning the same label to adjacent 'on' pixels within a column

kernel **vertLabel** (input: $I_\lambda \in \mathbb{Z}^{w\times h}$, output: $I_\lambda \in \mathbb{Z}^{w\times h}$) :
int col = $threadID$: $[0,\ w-1]$
for row from $h-2$ to 0 :
if($I^{(row,\lambda}$ and $I^{(row,\lambda}$ are both 'on')
then $I^{(row,\lambda}$

3.3 레이블 병합

열 단위 연속 화소 레이블 설정 작업이 끝나면 Fig. 6과 같은 결과를 얻게 된다. 모든 화소들이 가진 레이블은 해당 열 내에서 연결 객체를 대표하는 화소의 값이 된다. 알고리즘에 의해 이 값은 연결 객체 가운데 가장 아래쪽에 위치한 화소를 가리킨다. 우리는 이 대표 화소를 해당 연결 객체 화소들의 루트(root)라고 부를 수 있다.

문제는 대표 화소가 하나의 열(column)만을 고려하여 결정된 상태라는 것이다. Fig. 6의 경우 17번 화소는 왼쪽에 놓은 화소가 가리키는 56번 화소를 대표 화소로 변경되어야 한다. 이를 위해서는 좌우로 인접한 화소를 고려하여 인접한 화소가 자신의 대표 화소보다 더 큰 값을 갖는 경우 더 나은 대표 화소로 값을 변경하는 것이다.

이 작업은 하나의 스레드가 두 개의 열을 담당하여 두 열에서 각 행에 존재하는 두 개 씩의 화소를 비교한 뒤, 두 화소가 모두 켜져 있을 경우 둘의 대표 화소 가운데 큰 값을 두 화소 모두의 대표 화소로 설정하는 작업을 수행한다. 1 단계 수행이 끝나고 나

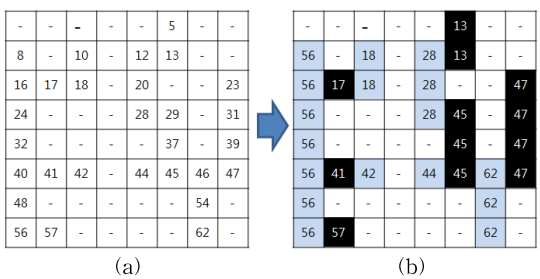


Fig. 6. Labels obtained after column-wise pixel run computation. (a) initial label image I_λ (b) column-wise label update.

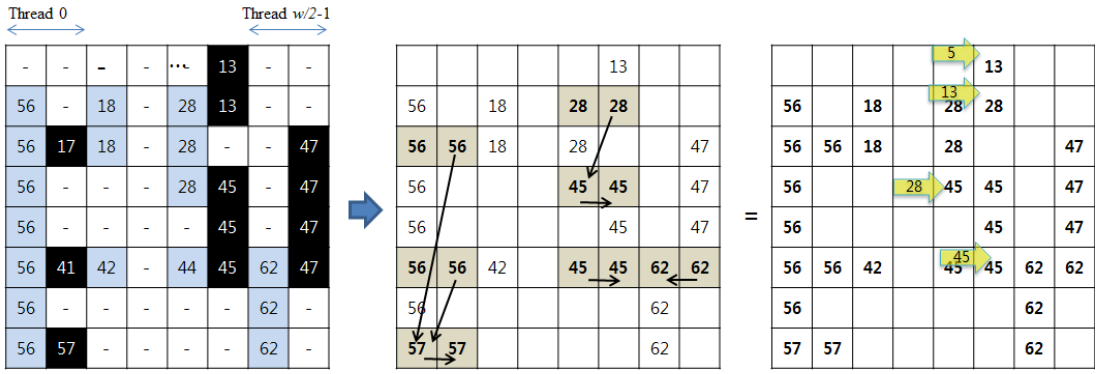


Fig. 7. Column-wise merge. (a) two adjacent columns are assigned to a single thread (b) two apixels in an identical row are compared and updated (c) column-wise update result.

면, 두 개의 열로 이루어진 $w/2$개의 쌍에 대한 레이블 통합이 이루어진다.

Fig. 7은 이러한 과정을 보이고 있다. 가장 왼쪽의 이미지는 열 단위로 연결 요소 레이블링이 마무린 된 상태이다. 이 이미지에 대해 두 개의 열을 묶어 레이블을 통합하는 작업을 수행한다. 각 행은 서로 독립적으로 다루어질 수 있으므로 다른 스레드가 담당한다. 따라서 최초의 작업은 $h \cdot w/2$ 개의 스레드가 필요하다. 각 스레드는 두 개의 열에서 동일한 행을 가진 화소 쌍을 비교한다. 비교의 대상은 두 화소에 기록된 대표 화소이다. 두 화소 모두 켜져 있는데, 두 화소의 대표 화소가 다른 경우 큰 값을 가진 대표 화소로 일치시킨다. 이러한 작업의 결과가 Fig. 7의 가운데 있는 이미지이다. 이 이미지는 대표 화소가 변경된 화소가 어떤 화소를 대표로 삼아 변경되었는지도 보이고 있다. 화살표가 표시된 화소들이 변경된 화소이다.

이때 어떤 화소의 소속은 대표 화소 번호로 연결된 트리의 루트(root)로 알 수 있다. Fig. 7의 가장 오른쪽 이미지를 보자. 예를 들어 이미지의 5번 화소를 보자. 5번 화소에 레이블링된 대표 화소 값은 13이다. 13번 화소를 따라가면 대표화소가 28임을 알 수 있다. 28번째 화소에 가서 대표화소를 보면 다시 45로 기록되어 있다. 45번 화소는 대표 화소가 45이다. 즉 자신이 현재까지 발견된 계층구조(hierarchy)에서 루트임을 의미한다. 따라서 5번 화소는 45번화

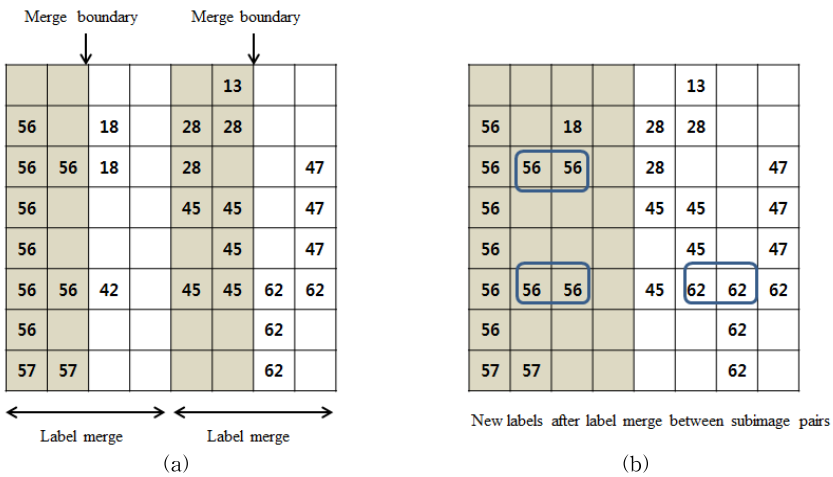


Fig. 8. Label merge between subimage pairs. (a) ranges and boundaries of merge tasks (b) result of label merge between subimage pairs.

소를 루트로 하는 화소 그룹이 만들어내는 연결요소의 원소임을 알 수 있다.

앞서 설명한 방식은 두 개의 열로 이루어진 부이미지(subimage)에서의 레이블 통합이다. 이 단계를 마치면 두 개의 부이미지를 통합하여 더 큰 부이미지를 만들어야 한다. Fig. 8의 (a)는 Fig. 7의 단계를 마치고 4 개의 부이미지로 구성된 레이블 통합 중간 단계이다. 다음 단계는 부이미지 두 개씩을 통합하는 것이고, 통합에 필요한 비교는 통합되는 부이미지의 경계에서만 이루어진다. 이를 통합 경계로 표현했다. 통합 방식은 앞서 설명한 방법과 동일하며 Fig. 8의 (b)는 통합 결과이다. 부이미지 2개씩 통합되었으므로 원래의 부이미지 개수가 k개였다면 $k/2$개의 부이미지를 얻게 된다. Fig. 8의 (b)는 두 장의 부이미지가 회색과 백색으로 구분되어 표현되었다. 레이블이 변경된 부분은 사각형으로 묶어서 보이고 있다.

이러한 레이블 통합은 전체 이미지가 하나의 부이미지로 통합될 때까지 반복된다. 이 반복의 회수는 이미지의 너비 w에 대해 $\log_2 w$가 됨을 쉽게 알 수 있다. 이러한 통합은 알고리즘 3과 같은 방식으로 구현된다. 이때 다소 까다로운 문제가 있다. Fig. 9를 살펴보자. 왼쪽의 통합전 상황을 보면 10번째 화소는 대표화소가 18 번이고, 18번 화소는 대표가 19, 19 번 화소는 20 번 화소로 연결되어 있다. 그런데 레이블 통합 경계에서 인접한 화소를 비교하여 단순히 큰 값으로 변경하면 Fig. 9의 오른쪽 이미지를 얻게 된다. 여기서 10번 화소는 대표 화소 18을 가지고 18번 화소를 따라간다. 그런데 18번 화소의 대표 화소는 56으로 변경되어 19, 20 번 화소와의 연결이 끊어지는 결과가 발생한다.

이러한 문제를 해결하기 위해 비교 대상 화소의 레이블을 단순 병합하지 않고, 해당 화소가 속한 연결요소 그룹의 루트 노드를 찾아 루트 노드의 레이블을 비교한다. 그리고 경계에 붙어 비교의 대상이 되었던 화소가 아니라 루트 노드의 레이블을 변경하는 방식으로 레이블 병합을 수행한다. Fig. 10에 이러한 방법에 대한 예시가 있다. 비교의 대상이 되는 두 화소는 56과 19라는 대표 화소를 가지고 있다. 56번 화소에 가면 이 화소는 57번 화소를 대표 노드로 하고 있으며, 57번 노드가 이 그룹의 루트임을 알 수 있다. 19번 화소를 대표 노드로 하는 화소는 이 연결을 따라가면 20번 화소가 루트임을 알 수 있다. 따라서 비교 대상 화소 둘이 속한 그룹의 루트는 각각 57과 20이다. 이 둘 중에 루트 역할을 수행할 수 있는 화소는 57번 화소이다. 따라서 20번 노드의 레이블을 그림과 같이 57로 변경한다. 이렇게 하면 연결성 훼손없이 부이미지의 화소들을 속한 그룹 내에서 계층구조를 가진 트리로 병합해 나갈 수 있다.

처음에는 각각의 열을 부이미지로 간주하므로 모

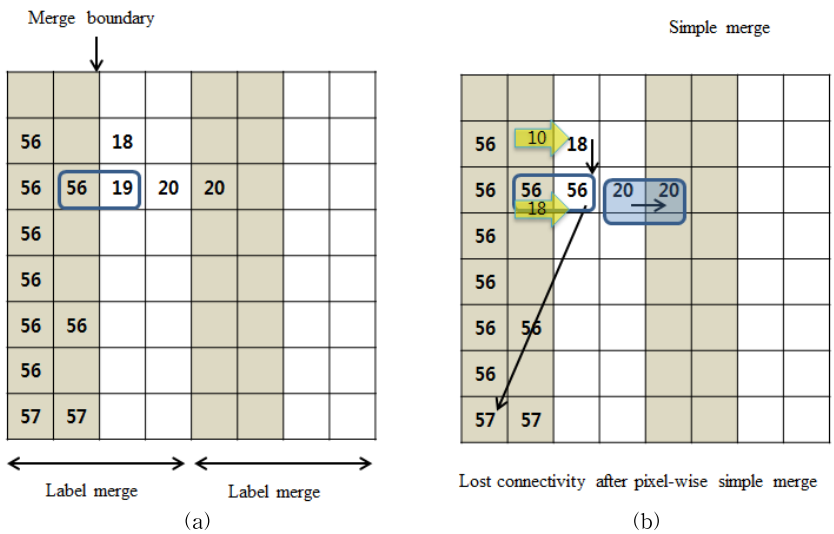


Fig. 9. Connectivity lost problem when pixel-wise simple label merge is performed. (a) two pixels are to be compared (b) simple merge with connectivity loss.

두 w개의 부이미지가 있다. 이들 부이미지들을 2개씩 묶어 병합할 때 경계가 되는 곳은 $w/2$ 군데이며, 처음 스레드의 개수는 바로 이 $w/2$개에 이미지를 구성하는 행의 개수 h가 곱해진 $h \cdot w/2$가 된다. 그 다음 단계에서 부이미지의 수는 반으로 줄어들며, 이에 따라 다음 병합 경계는 $h \cdot w/2^2$이며, k 번째 병합 단계에서 병합 대상이 되는 경계는 $h \cdot w/2^k$개가 된다. 따라서 이러한 병합작업은 $\log_2 w$ 번 반복된다.

Fig. 11은 이러한 레이블 갱신 방법을 개념적으로 보이고 있다. Fig. 11의 (a)는 두 개의 연결 요소를 구성하는 화소 각각이 가진 대표 화소 레이블을 계층구조로 가시화한 결과이다. 레이블 병합 과정에서 Fig. 11의 (b)와 같이 새로이 발견되는 연결 화소 쌍이 있다. 이 두 화소의 레이블을 Fig. 11의 (c)와 같

Algorithm 3. host and kernel function for hierarchical merge of labels

host **callMergeLabel**(input: $I_\lambda \in \mathbb{Z}^{w \times h}$, output: $I_\lambda \in \mathbb{Z}^{w \times h}$) :
    int div = 2
    for i from 0 to $\log_2 w - 1$:
        mergeLabel<<number of threads = $h \cdot w/\text{div}$>>(div, I_λ)
device **findRoot**(input: row, col $\in \mathbb{Z}$, output: label $\in \mathbb{Z}$) :
    if $I_\lambda^{(row,col)}$ is 'off' return -1
    int idx = $w \cdot row + col$
    while ($I_\lambda^{idx} \neq w \cdot row + col$) idx = I_λ^{idx}
    return idx
kernel **mergeLabel**(input: div $\in \mathbb{Z}$, $I_\lambda \in \mathbb{Z}^{w \times h}$, output: $I_\lambda \in \mathbb{Z}^{w \times h}$) :
    threadID : [0, $h \times w/\text{div} - 1$]
    int nColumnCheck = w/div
    int col = div/2 + (*threadID* % nColumnCheck)*div
    int row = *threadID* / nColumnCheck
    if ($I_\lambda^{(row,col)}$ and $I_\lambda^{(row,col+1)}$ are both 'on') :
        I_λ^{lRoot} = **findRoot** (row, col)
        I_λ^{rRoot} = **findRoot** (row, col+1)
        m = min(I_λ^{lRoot}, I_λ^{rRoot})
        M = max(I_λ^{lRoot}, I_λ^{rRoot})
        I_λ^{m} = I_λ^{M}

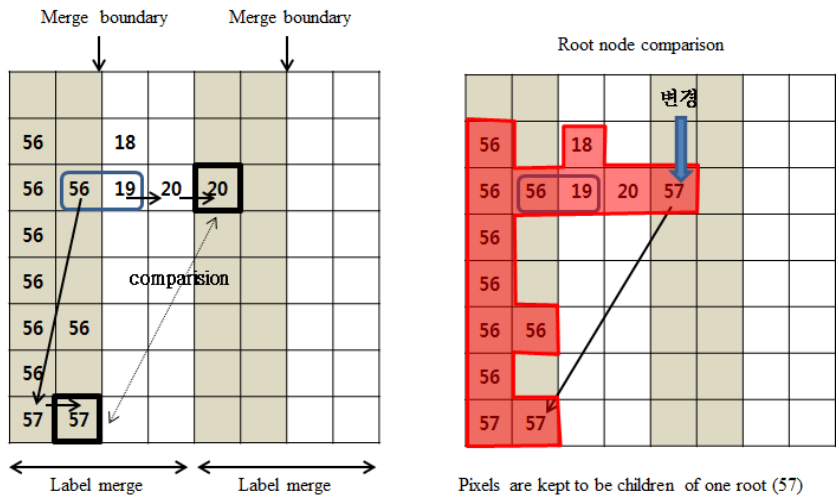


Fig. 10. Pixels are compared by the root of each node and the label update is performed on one of the roots in order to keep the connectivity.

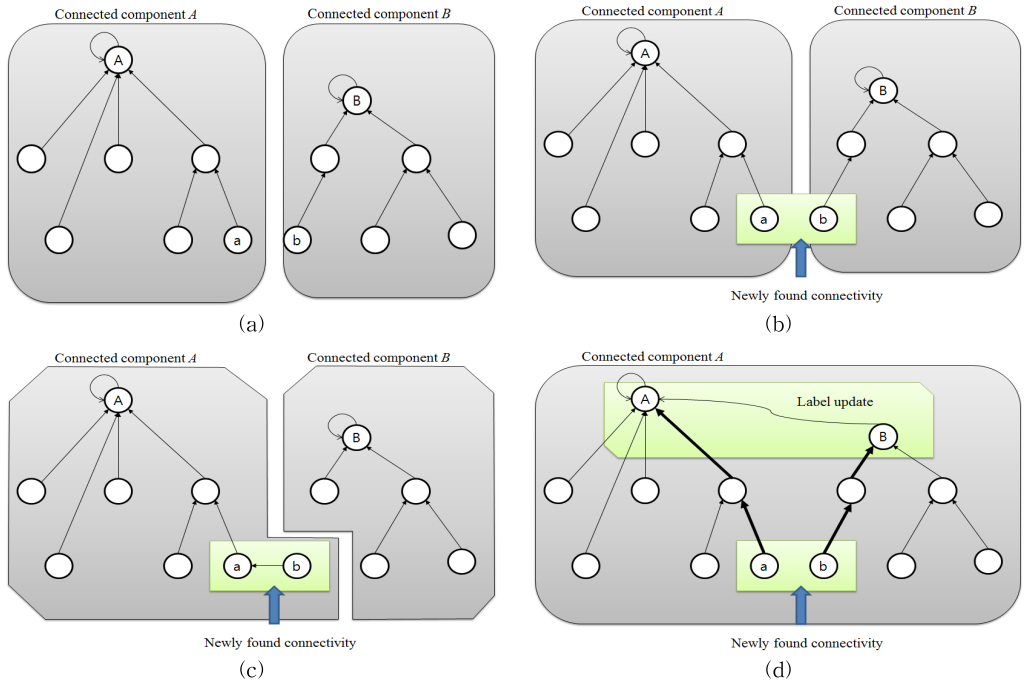


Fig. 11. The concept of label update with consideration of newly found connectivity. (a) two connectivity hierarchies (b) newly found connectivity (c) lost connectivity by pixel-wise label update (d) preserved connectivity after root node comparison.

이 갱신할 경우 두 연결요소는 여전히 떨어져 있게 된다. 이를 회피하기 위해서는 Fig. 11의 (d)와 같이 두 화소의 루트를 찾아 루트들 사이의 레이블을 비교하고 갱신하는 작업이 필요하다.

3.4 레이블 재지정

Fig. 10에서 볼 수 있는 바와 같이 레이블 병합을 통해 얻어진 결과는 각 화소가 대표 화소로 연결된 계층 구조, 즉 트리 구조를 갖게 된다. 우리의 최종 목표는 하나의 연결 요소가 하나의 레이블을 갖게 하는 것이다. 이를 위해 레이블을 재지정하는 방법이 필요하다. 이것은 이미 살펴본 알고리즘 3에 나타나 있는 findRoot 함수를 활용하여 각 픽셀이 자신이 속한 그룹의 루트 화소를 찾아 그 값으로 갱신하면 된다. 이는 알고리즘 4와 같이 커널 함수를 간단히 구현하여 모든 화소의 수만큼 스레드를 생성한 뒤에 병렬적으로 처리할 수 있다.

3.5 연결요소 개수 구하기

레이블 지정이 끝나면 이미지 내에 연결요소가 모두 몇 개인지를 세는 작업이 필요하다. 이것은 루트 역할을 수행하는 화소가 몇 개인지를 확인하면 된다. 이를 확인하기 위해서는 모든 화소의 개수인 $w \cdot h$ 개의 스레드를 생성하고, 각 스레드가 담당하는 화소마다 가지고 있는 대표화소 레이블이 자기 자신을 가리키는지를 확인하면 된다. 자기 자신을 가리킬 경우 해당 화소는 루트 화소이기 때문이다. 이렇게 각각의 스레드가 담당한 화소가 루트 노드인지를 확인한 뒤, 루트 노드인 경우 연결요소의 개수를 저장하고 있는 변수를 증가시키면 된다. 여기서

Algorithm 4. Relabeling for assigning an identical label for connected pixels

kernel **relabel** (input: $I_\lambda \in \mathbb{Z}^{w \times h}$, output: $I_\lambda \in \mathbb{Z}^{w \times h}$) :
$threadID$: [0, $w \times h - 1$]
if ($I_\lambda^{threadID}$ is 'on') $I_\lambda^{threadID}$ = **findRoot**(row=$threadID/w$, col=$threadID\%w$)

주의할 점은 루트인 화소를 담당한 스레드가 변경하는 변수는 임계구역(critical section)이 된다는 것이다. 이를 단순히 다음과 같이 구현할 경우 잘못된 계수 결과를 얻게 된다.

if ($I_{\lambda}^{threadID}$ = $threadID$) then $c \leftarrow c+1$

따라서 CUDA에서 제공하는 atomicAdd와 같이 임계구역 변수를 적절히 다룰 수 있는 방법을 사용하여 이 변수를 증가시켜야 한다. 알고리즘 5는 이를 고려하여 구현된 연결요소 개수 계수 방법이다.

Algorithm 5. Counting the number of connected components

```
kernel countBlobs (input: I_λ ∈ Z^{w×h}, output:
nBlobs ∈ Z) :
    threadID : [0, w×h−1]
    if ( I_λ^{threadID} = threadID ) atomicAdd(c, 1)
```

3.5 연결요소 바운딩 박스 계산

연결요소의 개수 n이 얻어지고 나면 각각의 연결요소가 가진 바운딩 박스 등을 계산하는 일이 필요하다. 이를 위해서 n개의 바운딩 박스 객체가 생성되고, 각각의 화소는 자신이 속한 연결요소에 해당하는 바운딩 박스의 정보를 갱신하게 된다. n개의 바운딩 박스를 준비한다. 한 개의 바운딩 박스는 x축 최소값, x축 최대값, y축 최소값과 y축 최대값을 표현하는 4개의 값을 가진다. i번째 연결요소를 표현하는 바운딩 박스 b_i가 가진 이 4개의 값을 $b_{i,0}$, $b_{i,1}$, $b_{i,2}$, $b_{i,3}$으로 표현하자. n개의 바운딩 박스는 정수를 원소로 하는 $\mathbb{Z}^{n\times 4}$의 행렬로 표현할 수 있다.

이제 스레드를 $w \cdot h$개 생성하고 각각의 스레드가 하나의 화소를 담당한다. 이 화소가 몇 번째 연결 요소인지는 레이블을 통해 쉽게 확인할 수 있다. 이 화소의 좌표가 (x,y)라고 하고, k번째 연결 요소라고 하면 다음과 같이 바운딩 박스 갱신을 하면 된다.

$$\begin{aligned} x < b_{k,0} &\rightarrow b_{k,0} := x \\ x > b_{k,1} &\rightarrow b_{k,1} := x \\ y < b_{k,2} &\rightarrow b_{k,2} := y \\ y < b_{k,3} &\rightarrow b_{k,3} := y \end{aligned} \tag{2}$$

그런데 이러한 작업이 모든 화소별로 병렬적으로 이루어지기 때문에 바운딩 박스가 임계 영역으로 취급되어야 한다는 것을 쉽게 알 수 있다. 그런데,

Algorithm 6. Obtaining bounding boxes considering them critical sections

```
kernel extractBounds (input: I_λ ∈ Z^{w×h}, b ∈ Z^{n×4},
    l ∈ Z^n, output: b ∈ Z^{n×4}) :
    threadID : [0, w×h−1]
    if ( I_λ^{threadID} = 'on' ) :
        int k = whichGroup(I_λ^{threadID})
        (x,y) = (threadID%w, threadID/w)
        bool waiting = true
        while(waiting) :
            if(atomicExch(l_k, 1) == 0) :
                if x < b_{k,0} then b_{k,0} = x
                if x > b_{k,1} then b_{k,1} = x
                if y < b_{k,2} then b_{k,2} = y
                if y > b_{k,3} then b_{k,3} = y
                atomicExch(l_k, 0)
                waiting=false
```

CUDA에서는 임계영역을 다루기 위해 메모리 잠금을 할 수 있는 기능이 없기 때문에 직접 메모리 잠금을 구현해야 한다. 이것은 어떤 화소가 k 번째 연결요소이면서 b_k를 갱신하고 있을 때, 다른 k 연결요소에 포함된 어떤 다른 화소도 b_k를 갱신하지 못 하고 대기해야 함을 의미한다. 이것은 상당한 계산 병목이 될 수 있어 조심스럽게 구현되어야 한다.

메모리 잠금을 구현하기 위해 atomicExch(v, v^{new})를 사용할 수 있다. 이 함수는 v를 v^{new}로 바꾸는 작업을 수행하면서 원래 저장되어 있던 v를 리턴한다. 이 과정에서 v를 임계영역으로 다룬다. 즉, 바운딩 박스 b_k를 변경할 때, 다른 스레드가 b_k를 갱신하지 않고 기다리도록 만드는 방법은 해당 바운딩 박스를 접근할 수 있는지 없는지를 알려주는 잠금 변수 $l \in \mathbb{Z}^n$를 생성한 뒤 이 atomicExch를 이용하여 접근권한을 제어한다.

이 잠금변수 가운데 l_k가 0이면 k 번째 바운딩 박스 b_k에 접근할 수 있다. 이때는 다른 스레드가 접근하지 못 하도록 atomicExch를 이용하여 1로 바꾸고 접근한다. 사용이 끝나면 이를 다시 0으로 바꾼다. 만약 같은 k 번 바운딩 박스를 접근해야 하는 스레드가 있다면 대기상태가 되고, l_k가 0이 될 때까지 계속해서 무한루프를 돌며 접근권한을 얻을 때까지 기다린다. 알고리즘 6이 이를 구현한 것이다. 이때 whichGroup 함수는 특정한 레이블을 가진 화소가

어떤 그룹 k에 속하는지를 알려주는 함수이다.

그런데 이 기법은 어떤 화소가 $b_{k,0}$을 갱신하고 있을 때, 다른 어떤 화소가 $b_{k,1}$을 갱신하는 것까지 막게 된다. 이를 해결하기 위해서는 바운딩 박스 단위가 아니라 바운딩 박스에서 관리되는 4 개의 좌표 최대값과 최소값에 대한 접근권한을 따로 관리하는 잠금변수를 생성하는 것이 좋을 것이다. 이제 l은 $l \in \mathbb{Z}^{n\times 4}$가 되며, 알고리즘 6은 알고리즘 7과 같이 개선될 수 있다.

Algorithm 7. Component-wise locking for efficient bounding box computation

```
kernel extractBounds (input: I_λ ∈ Z^{w×h}, b ∈ Z^{n×4},
      l ∈ Z^n, output: b ∈ Z^{n×4}) :
   threadID : [0, w×h−1]
   if ( I_λ^{threadID} = 'on' ) :
      int k = whichGroup(I_λ^{threadID})
      (x,y) = (threadID%w, threadID/w)

      while(x<b_{k,0}) :
         if(atomicExch(l_{k,0}, 1) == 0) :
            b_{k,0} = x , atomicExch(l_{k,0}, 0)
      while(x>b_{k,1}) :
         if(atomicExch(l_{k,1}, 1) == 0) :
            b_{k,1} = x , atomicExch(l_{k,1}, 0)
      while(y<b_{k,2}) :
         if(atomicExch(l_{k,2}, 1) == 0) :
            b_{k,2} = y , atomicExch(l_{k,2}, 0)
      while(y>b_{k,2}) :
         if(atomicExch(l_{k,3}, 1) == 0) :
            b_{k,3} = y , atomicExch(l_{k,3}, 0)
```

알고리즘 6과 7은 상당한 성능 차이를 보인다. 실험을 통해 알고리즘 6이 알고리즘 7보다 여섯 배 이상의 계산 시간을 요구하는 것을 확인할 수 있다.

4. 실험 결과 및 고찰

본 논문의 기법은 일반적인 수준의 CPU와 GPU를 장착한 컴퓨터에서 구현되었다. 실험 결과 데이터를 수집하기 위해 구현된 시스템은 i7-3630QM 2.4GHz CPU와 Geforce GTX 670MX GPU, 8 기가 바이트의 RAM을 장착하였고, Window 7을 운영체제로 사용하였다.

Fig. 11은 4096 × 4096 화소 이미지로 표현된 레이더 데이터에서 본 논문의 기법을 통해 연결요소를 찾아낸 결과이다. 각각의 연결 요소는 바운딩 박스 정보가 함께 추출되었고, 레이더 수신국에서의 거리, 연결 요소의 방향 등을 얻을 수 있도록 구현되었다.

Table 1은 제안된 기법을 4096 × 4096 사이즈의 레이더 신호 한 프레임에 적용하여 연결 요소를 추출할 때에 소요된 시간을 측정한 결과이다. 각각의 시간들은 100회의 수행 결과를 평균한 값이다. 실험은 두 가지 경우를 수행하였다. 하나는 50 개의 연결 요소가 있는 경우이며, 다른 하나는 1869 개의 많은 연결요소가 있는 경우이다. 가장 첫 행에는 가우시안 필터를 적용하여 잡음을 제거하는 데에 소요된 시간이 나타나 있다. 두 경우 모두 같은 시간 16.05 밀리초가 소요되었다. 잡음제어는 이웃 노드를 살펴보고 각자의 픽셀 강도를 조정하는 것으로 완벽히 병렬 처리로 수행할 수 있다. 따라서 연결 요소가 늘어도 시간의 차이가 없는 것이 당연하다. 이것은 CCL을 완벽히 병렬화하였을 때에 얻을 수 있는 성능의 한계라고 볼 수 있다.

연결 요소 레이블링(CCL)은 레이블을 부여하는 것과 바운딩 박스를 추출하는 두 가지 단계를 나누어 분석하였다. 전체적으로 레이블의 수가 많아지면 계산시간도 자연스럽게 더 필요하였다. 하지만 레이블 부여 자체만 보면 50 개 연결 요소의 경우 16.93 밀리초, 1869 개의 연결 요소인 경우 17.44 밀리초가 소요되어 큰 차이를 보이지 않음을 알 수 있다. 이것은 제안된 기법이 병렬 컴퓨팅 환경을 매우 잘 활용하고 있음을 보이는 것이다.

연결 요소의 개수가 크게 증가했을 때, 계산 시간이 급격히 늘어나는 부분은 바운딩 박스를 추출하는 부분이었다. 표에서 확인할 수 있는 바와 같이 알고리즘 7을 이용할 경우 알고리즘 6에 비해 효율적으로 추출을 할 수 있기는 하지만, 레이블이 늘어남에 따라 계산시간도 크게 늘어나는 것을 피하지는 못한다는 것을 확인할 수 있다. 그럼에도 불구하고 4096 × 4096 크기의 대규모 이미지 내에서 1986 개의 많은 연결요소를 찾는 데에도 바운딩 박스 처리를 포함하여 50 밀리초 이내에 처리할 수 있는 성능을 보이고 있다. 이는 레이더 신호를 다루는 데에 충분한 성능이다.

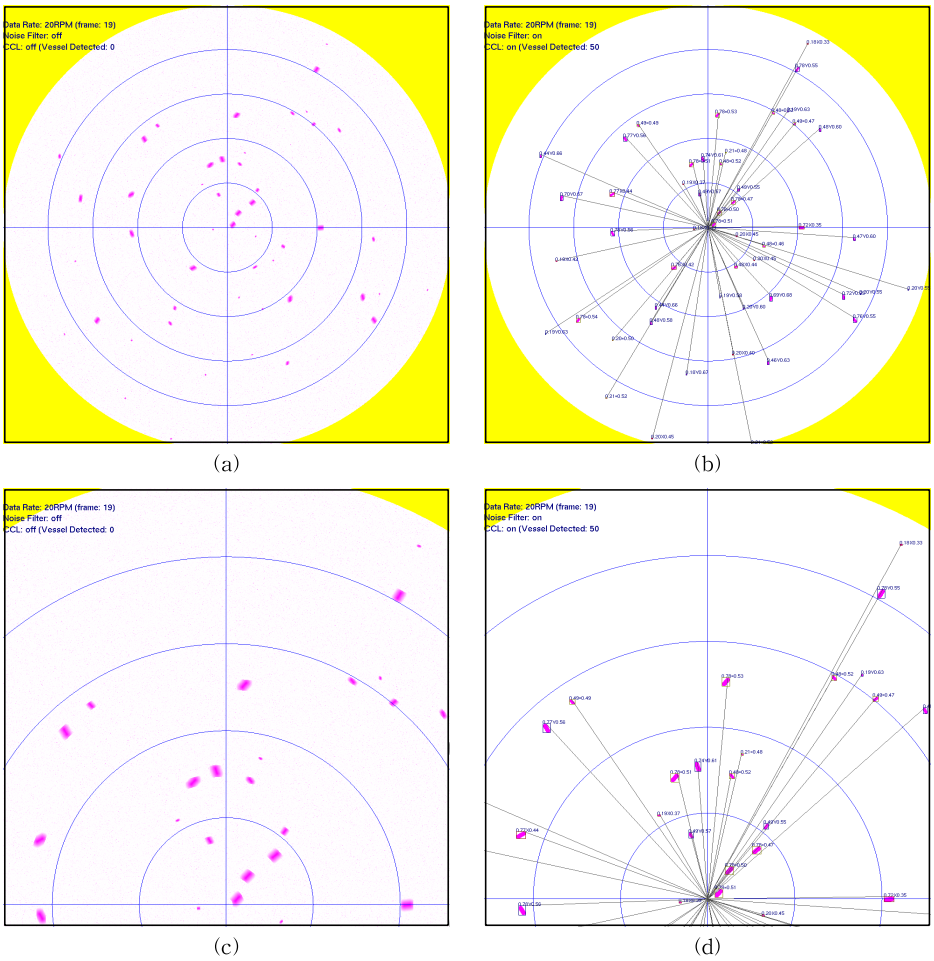


Fig. 12. The vessel tracking with proposed CCL techniques. (a) received radar signal (b) extracted connected components (c) zoom-in image of received radar (d) zoom-in image of vessel tracking image

Table 1. Performance analysis of GPU-based connected component labeling

Number of Connected Components		50		1869	
		Algorithm 6. Applied	Algorithm 7. Applied	Algorithm 6. Applied	Algorithm 7. Applied
Denoising with Gaussian filter		**16.05 msec**		**16.05 msec**	
CCL	label initialization	2.30 msec		2.29 msec	
	column-wise label merge	3.59 msec		3.63 msec	
	hierarchical label merge	7.97 msec		8.35 msec	
	relabeling	3.07 msec		3.17 msec	
	labeling	**16.93 msec**		**17.44 msec**	
	counting connected components	2.12 msec		2.16 msec	
	bounding box computation	15.76 msec	2.55 msec	117.55 msec	32.71 msec
	bounding box	**17.88 msec**	**4.67 msec**	**119.71 msec**	**34.87 msec**
total elapsed time for CCL		**34.81 msec**	**21.60 msec**	**137.15 msec**	**52.31 msec**

5. 결 론

본 논문에서 효율적인 연결 요소 레이블링(connected component labeling) 기법이 제안되었다. 제안된 기법은 GPU의 병렬처리 능력을 활용하여 구현되었다. 연결 요소 레이블링 기법은 다양한 영상처리 분야에서 핵심적으로 요구되는 기초 기술이며, 영상 처리의 성능을 좌우하는 중요한 연산이다. 특히 대용량의 이미지가 분석되어야 하는 응용분야에서는 데이터를 전처리하기 위해 효율적인 CCL 기법을 활용하는 것이 매우 중요하다. 본 논문은 특히 일반적인 이미지보다 해상도가 매우 큰 영상으로 처리되는 레이더 신호에 대해 효율적인 실시간 분석을 할 수 있도록 GPU 기반의 빠른 연결 요소 추출 기법을 제안하였다. 제안된 기법은 레이블을 부여하는 작업에서는 간단한 작업인 잡음제거 수준의 높은 성능을 보였으며, 연결요소가 늘어도 계산량이 크게 늘지 않아 레이더 신호 분석 등에 성공적으로 적용될 수 있다.

REFERENCE

[1] P. Chen, H.L. Zhao, C. Tao, and H.S. Sang. “Block-run-based Connected Component Labelling Algorithm for GPGPU Using Shared Memory,” *Electronics Letters*, Vol. 47, No. 24, pp. 1309-1311, 2011.

[2] O. Stava and B. Benes, *Connected Component Labeling in CUDA*, GPU Computing Gems, Morgan Kaufmann, Burlington, Massachusetts, 2011.

[3] H.L. Zhang, H.S. Sang, T.X. Zhang, and Y.B. Fan, “‘Line-based Cascade Labeling Algorithm for Hyper-scale Issue,” *International Conference on Multimedia Technology*, pp. 572-575, 2011.

[4] L. He, Y. Chao, and K. Suzuki. "A Run-based Two-scan Labeling Algorithm," *IEEE Transactions on Image Processing,* Vol. 17, No. 5, pp. 749-756, 2008.

[5] M. Minsky and S. Papert, *Perceptron*. MIT Press, Cambridge, 1969.

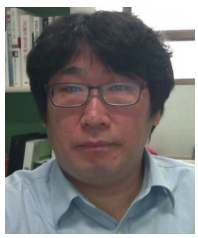
[6] H.M. Alnuweiri and V.K. Prasanna. "Parallel Architectures and Algorithms for Image Component Labeling," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Vol. 14, No. 10, pp. 1014-1034, 1992.

[7] C. Grana, D. Borghesani, and R. Cucchiara, “Optimized Block-based Connected Components Labeling with Decision Trees,” *IEEE Transactions on Image Processing*, Vol. 19, No. 6, pp. 1596-1609, 2010.

[8] B. Preto, F. Birra, A. Lopes, and P. Medeiros, "Object Identification in Binary Tomographic Images Using GPGPUs," *International Journal of Creative Interfaces and Computer Graphics,* Vol. 4, No. 2, pp. 40-56, 2013.

[9] S. Zavalishin, I. Safonov, Y. Bekhtin, and I. Kurilin, "Block Equivalence Algorithm for Labeling 2D and 3D Images on GPU," *Electronic Imaging*, No. 2, pp. 1-7, 2016.

[10] Heewon Kye, “Fast Medical Volume Decompression Using GPGPU,” *Journal of Korea Multimedia Society*, Vol. 15, No. 5, pp. 624-631, 2012.



강 영 민

1996년 2월 부산대학교 전자계산학과 학사
1999년 2월 부산대학교 전자계산학과 석사
2003년 2월 부산대학교 전자계산학과 박사

2002년 1월 ~ 2002년 7월 제네바대학 미라랩 임시연구원
2003년 10월 ~ 2005년 3월 한국전자통신연구원 연구원
2005년 3월 ~ 현재 동명대학교 게임공학과 정교수
관심분야 : 컴퓨터 그래픽스, GPU 병렬화, 인공지능