(19) 대한민국특허청(KR)
(12) 공개특허공보(A)

(11) 공개번호 10-2021-0126913
(43) 공개일자 2021년10월21일

(51) 국제특허분류(Int. Cl.)
H04N 19/50 (2014.01) *H04N 19/119* (2014.01)
H04N 19/137 (2014.01) *H04N 19/182* (2014.01)
H04N 19/42 (2014.01)
(52) CPC특허분류
H04N 19/50 (2015.01)
H04N 19/119 (2015.01)
(21) 출원번호 10-2020-0044581
(22) 출원일자 2020년04월13일
심사청구일자 2020년04월13일

(71) 출원인
인천대학교 산학협력단
인천광역시 연수구 아카데미로 119 (송도동)
(72) 발명자
전광길
인천광역시 연수구 해송로 70, 209동 805호(송도동, 송도웰카운티2단지)
(74) 대리인
특허법인충정

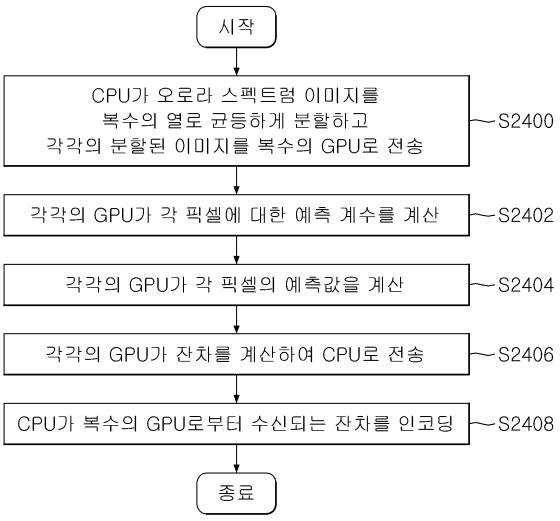
전체 청구항 수 : 총 5 항

(54) 발명의 명칭 **오로라 스펙트럼 데이터 압축 방법**

(57) 요 약

본 발명의 일 실시예에 의한 오로라 스펙트럼 데이터 압축 방법은, 그래픽 처리 유닛(GPU: Graphics Processing Unit)이, 오로라 스펙트럼 이미지의 각 픽셀에 대한 예측 계수를 계산하는 단계; 중앙 처리 장치(CPU)가, 상기 예측 계수에 기반하여 상기 각 픽셀의 예측값을 계산하는 단계; 상기 중앙 처리 장치(CPU)가, 상기 오로라 스펙트럼 이미지의 각 픽셀의 원래 값에서 상기 각 픽셀의 예측값을 감산하여 잔차를 획득하는 단계; 및 상기 중앙 처리 장치(CPU)가, 상기 잔차를 인코딩하는 단계를 포함하고, 상기 각 픽셀에 대한 예측 계수를 계산하는 단계는, 현재 픽셀 이전의 각 행의 복수의 픽셀들을 사용하여 복수의 선형 수학식을 형성하고, 상기 복수의 선형 수학식에 기반하여 상기 예측 계수를 계산한다.

대 표 도 - 도24



(52) CPC특허분류
H04N 19/137 (2015.01)
H04N 19/182 (2015.01)
H04N 19/42 (2015.01)

명 세 서

청구범위

청구항 1

중앙 처리 장치(CPU)가, 오로라 스펙트럼 이미지를 복수의 열로 균등하게 분할하고 각각의 분할된 이미지를 복수의 그래픽 처리 유닛(GPU: Graphics Processing Unit)으로 전송하는 단계;

각각의 그래픽 처리 유닛(GPU)이, 분할된 이미지의 각 픽셀에 대한 예측 계수를 계산하는 단계;

상기 각각의 그래픽 처리 유닛(GPU)이, 상기 예측 계수에 기반하여 상기 각 픽셀의 예측값을 계산하는 단계;

상기 각각의 그래픽 처리 유닛(GPU)이, 상기 분할된 이미지의 각 픽셀의 원래 값에서 상기 각 픽셀의 예측값을 감산하여 잔차를 획득하고, 상기 획득된 잔차를 상기 중앙 처리 장치(CPU)로 전송하는 단계; 및

상기 중앙 처리 장치(CPU)가, 상기 복수의 그래픽 처리 유닛(GPU)으로부터 수신된 복수의 잔차를 인코딩하는 단계를 포함하고,

상기 각 픽셀에 대한 예측 계수를 계산하는 단계는, 현재 픽셀 이전의 각 행의 복수의 픽셀들을 사용하여 복수의 선형 수학식을 형성하고, 상기 복수의 선형 수학식에 기반하여 상기 예측 계수를 계산하는, 오로라 스펙트럼 데이터 압축 방법.

청구항 2

청구항 1에 있어서,

상기 각 픽셀($x_{m,n}$)에 대한 예측 계수는 예측 계수 벡터($\begin{bmatrix} \alpha_0 & \alpha_1 & \cdots & \alpha_N \end{bmatrix}^T$)로부터 획득되고, 상기 예측 계수 벡터는 수학식 5에 기반하여 계산되며,

[수학식 5]

$$\begin{bmatrix} x_{0,n-N-M+1} & x_{0,n-N-M+2} & \cdots & x_{0,n-M} \\ x_{0,n-N-M+2} & x_{0,n-N-M+3} & \cdots & x_{0,n-M+1} \\ \vdots & \vdots & \cdots & \vdots \\ x_{0,n-N} & x_{0,n-N+1} & \cdots & x_{0,n-1} \\ \vdots & \vdots & \cdots & \vdots \\ x_{m-1,n-N-M+1} & x_{m-1,n-N-M+2} & \cdots & x_{m-1,n-M} \\ x_{m-1,n-N-M+2} & x_{m-1,n-N-M+3} & \cdots & x_{m-1,n-M+1} \\ \vdots & \vdots & \cdots & \vdots \\ x_{m-1,n-N} & x_{m-1,n-N+1} & \cdots & x_{m-1,n-1} \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_N \end{bmatrix} = \begin{bmatrix} x_{0,n-M+1} \\ x_{0,n-M+2} \\ \vdots \\ x_{0,n} \\ \vdots \\ x_{m-1,n-M+1} \\ x_{m-1,n-M+2} \\ \vdots \\ x_{m-1,n} \end{bmatrix},$$

$x_{i,j}\ (i = 0, 1, \cdots, m-1,\ j = n-N, n-N+1, \cdots, n)$는 i번째 행과 j번째 열의 픽셀이고, N은 예측 차수로서 예측 계수들을 계산하는데 사용된 이전의 열 수를 나타내며, M은 $x_{m,n}$ 이전의 행들 각각으로 구성된 수학식들의 수인, 오로라 스펙트럼 데이터 압축 방법.

청구항 3

청구항 1에 있어서,

상기 잔차를 인코딩하는 단계는, 잔차 발생 횟수가 소정 범위 내에 있는 잔차만 인코딩하고, 상기 잔차 발생 횟수가 상기 소정 범위를 벗어나는 잔차는 인코딩하지 않고 그대로 출력하는 단계를 포함하는, 오로라 스펙트럼 데이터 압축 방법.

청구항 4

청구항 2에 있어서,

상기 수학식 5에서 좌변의 첫 번째 행렬을 C로 표시하고 예측 계수 벡터 $\begin{bmatrix} \alpha_0 & \alpha_1 & \cdots & \alpha_N \end{bmatrix}^T$ 를 $\vec{\alpha}$로 표시하며 우변의 벡터를 $\vec{b}$로 표시하면, 상기 수학식 5는 $C\vec{\alpha} = \vec{b}$ 로 표시되며, 상기 예측 계수 벡터($\vec{\alpha}$)는 수학식 2에 기반하여 계산되고,

[수학식 2]

$$\vec{\alpha} = (C^T C)^{-1} C^T \vec{b}$$

상기 행렬 C를 $C(x_{m,n})$ 으로 표시할 때, 상기 수학식 2에서 $C^T C$ 는 $C^T(x_{m,n})C(x_{m,n})$ 로 표시되며, $C^T(x_{m,n})C(x_{m,n})$ 는 수학식 6에 기반하여 계산되고,

[수학식 6]

$$C^T(x_{m,n})\,C(x_{m,n}) = C^T(x_{m-1,n})\,C(x_{m-1,n}) + C^T{}_{m-1,n} C_{m-1,n}$$

$C_{m-1,n}$ 은

$$C_{m-1,n} = \begin{vmatrix} x_{m-1,n-M-(N-1)} & \cdots & x_{m-1,n-M} \\ \vdots & & \vdots \\ x_{m-1,n-N} & \cdots & x_{m-1,n-1} \end{vmatrix}$$

인, 오로라 스펙트럼 데이터 압축 방법.

청구항 5

청구항 4에 있어서,

$C^T C$ 의 역행렬인 $(C^T C)^{-1}$ 은 가우스 조단 소거법(Gaussian Jordan Elimination Method)을 이용하여 계산되는, 오로라 스펙트럼 데이터 압축 방법.

발명의 설명

기 술 분 야

[0001] 본 발명은 오로라 스펙트럼 데이터 압축 방법에 관한 것이다.

배 경 기 술

[0002] 오로라는 자연에서 가장 아름다운 경이로 여겨지며 화려하고 끊임없이 변화한다. 우주에서 고 에너지로 하전된 입자가 태양풍에 의해 지구로 운반될 때, 지구의 극으로 지자기장에 의해 끌어당겨져, 대기권의 분자와 원자가 충돌하여 결국 오로라가 발생한다. 오로라는 매우 중요한 연구 가치를 가지고 있다. 그들은 태양 활동의 연구와 지구에 미치는 영향을 파악하는데 도움이 된다. 또한, 오로라 연구는 다른 행성을 연구하기 위한 참고 자료를 제공할 수 있는데, 이는 오로라가 태양계에서 자기장이 있는 행성에 흔한 현상이기 때문이다.

[0003] 오로라 스펙트럼 데이터는 극지방에 장착된 분광계에 의해 캡처된다. 오로라의 정확한 발생 시간을 알 수 없으므로, 매우 높은 주파수에서 오로라 스펙트럼 데이터를 연속적으로 캡처하려면 분광계가 필요하다. 결과적으로, 오로라 스펙트럼 데이터의 양이 매우 크기 때문에 저장 및 전송에 큰 어려움이 있다. 이 문제를 해결하기 위해, 오로라 스펙트럼 데이터의 압축은 필수적이며, 손실없는 압축이 손실있는 압축보다 선호된다. 여기에는 두 가지 주요 이유가 있다. 하나는 오로라 스펙트럼 이미지의 과학적 가치가 매우 높고 획득 비용이 상당히 비싸기 때문에 오로라 스펙트럼 이미지는 장기 보존 가치가 있다는 점이고, 다른 하나는 일부 응용에서 사소한 정보 손실이 큰 오류를 유발할 수 있다는 점이다.

[0004] DPCM(Differential Pulse Code Modulation)의 아이디어는 데이터를 선형 회귀 예측과 연관시키는 것이다. Aiazzi et al.은 거의 무손실 및 무손실 스틸 이미지 압축 알고리즘에 기반한 두 가지 선형 회귀 예측 기반을 제안했다. 하나는 RCP(Relaxation-Labelled Prediction)로 표시되며 이미지를 작은 블록으로 분할한 다음 각 블록의 선형 예측자를 계산한다. 다른 하나는 퍼지 논리 기반 매칭 추적(FMP)으로 명명되며, 퍼지 논리 기술을 통해 선형 회귀 예측 변수를 계산한다. 또한, Aiazzi et al.은 산술 코딩 전에 예측 오류를 동종 클래스로 분할하는 인과 관계 DPCM 방식에 대한 컨텍스트 기반의 거의 무손실 또는 무손실 코딩 방법을 제안했다. 그런 다음 Mielikainen과 Toivanen은 초 분광 이미지의 무손실 압축을 위해 C-DPCM (Clustered DPCM)이라는 방법을 제안했다. C-DPCM 알고리즘은 먼저 초분광 이미지의 모든 스펙트럼 라인을 여러 클래스로 클러스터링한 다음 최소 제곱법(LSM)을 사용하여 각 클래스의 각 스펙트럼 대역의 예측 계수를 계산한 후 잔차를 계산하고 마지막으로 범위 코더를 이용하여 예측 계수 및 잔차를 부호화한다. 범위 코더는 산술 코더와 유사하지만 산술 코더보다 거의 두 배 빠르다.

[0005] 초분광 및 오로라 스펙트럼 이미지는 모두 스펙트럼 데이터이므로 DPCM의 아이디어를 오로라 스펙트럼 이미지의 무손실 압축에 적용할 수 있다. 그러나, 오로라 스펙트럼 이미지는 1024개의 스펙트럼 밴드를 가지고 있는데, 이는 이전 연구에서 소개된 초분광 이미지의 220 또는 224 스펙트럼 밴드보다 훨씬 크다. 따라서 각 밴드의 예측 계수가 부호화되는 경우 전체 비트 전송률은 매우 클 것이다. 이 문제를 해결하기 위하여 본 발명에서는 온라인 압축 아이디어가 채택되었다. 온라인 압축의 주요 아이디어는 이미 코딩된 픽셀을 사용하여 현재 픽셀을 예측하는 것이므로 디코더는 이미 디코딩된 픽셀을 사용하여 현재 픽셀의 예측 계수를 계산할 수 있으며 예측 계수를 인코딩하여 전송할 필요가 없다. 결과적으로, 오로라 스펙트럼 데이터의 압축을 위해 온라인 DPCM을 사용할 때, 인코딩될 필요가 있고 디코더로 전송될 필요가 있는 유일한 데이터는 잔차이다. 또한 본 발명에서는 원래 온라인 DPCM 방법의 압축 성능을 향상시키기 위해 두 가지 개선을 제안했다. 하나는 예측 계수를 계산할 때 선형 수학식 시스템을 설정하고 다른 하나는 잔차를 인코딩하는 것이다.

[0006] DPCM 외에도 JPEG-LS(JPEG는 Joint Photographic Experts Group의 약어이며 JPEG-LS는 연속 톤 이미지의 무손실 및 거의 무손실 압축을 위한 새로운 표준이다)와 같은 다른 무손실 압축 알고리즘이 많이 존재한다. JPEG2000(JPEG2000은 뛰어난 압축 성능을 가진 JPEG의 후속 제품임) 및 CALIC(컨텍스트 기반, 적응형, 무손실 이미지 코덱)이다. JPEG-LS의 핵심 알고리즘은 LOw-COmplexity LOCO-I (LOssless Compression for Images) [10,11]로 고정 예측자가 수직 또는 수평 에지를 감지하는 데 사용되며 픽셀의 예측 값은 3개의 후보 예측값의 중앙값으로서 간주될 수 있다. JPEG2000은 웨이블릿 기반 스틸 이미지 압축 표준으로, 공통 프레임 워크 내에서 손실 및 무손실 압축을 모두 달성할 수 있다. CALIC은 많은 수의 모델링 컨텍스트를 사용하여 비선형 예측 변수를 조정하며, 비선형 예측 변수는 주어진 컨텍스트에서 과거 오류로부터 학습함으로써 스스로 적응할 수 있다. 그러나 JPEG-LS, JPEG2000 및 CALIC은 스펙트럼 데이터가 아닌 일반적인 이미지용으로 개발되었다.

[0007] 위에서 소개한 2차원(2-D) 압축 방법 외에도 몇 가지 3차원(3-D) 방법도 있다. Kim과 Pearlman은 손실 비디오 코딩을 위해 3D-SPIHT(계층 트리에서 3차원 세트 분할)를 제안했다. Lucas et al.은 의료 이미지의 무손실 압축을 위한 3D-MRP(Three-Dimensional Minimum Rate Predictors)라는 방법을 제안했다. Xiaolin과 Memon은 다중 스펙트럼 이미지의 무손실 압축을 위해 원본 CALIC을 3-D로 확장했다. 초분광 이미지는 일반적으로 매우 크고 적용 가치가 높기 때문에, 많은 연구자들은 3-D 방법을 사용하여 초분광 이미지 압축에 노력을 기울이고 있다. 실제로 시간 차원을 추가하면 오로라 스펙트럼 이미지를 3D 이미지로 간주할 수 있으며 3D 방법을 오로라 스펙트럼 이미지의 압축에 적용할 수도 있다. 그러나 남극 중산 스테이션은 중국과 거리가 멀고 전송 링크가 매우 제한되어 있으며 파일이 클수록 전송 오류가 발생할 가능성이 높으므로 오로라 스펙트럼 이미지는 일반적으로 프레임 단위로 압축되어 전송된다.

선행기술문헌

특허문헌

[0008] (특허문헌 0001) KR 10-2048340 B1

발명의 내용

해결하려는 과제

[0009] 본 발명이 해결하고자 하는 과제는 오로라 스펙트럼 데이터를 압축하는데 소요되는 시간을 최소화할 수 있는 오로라 스펙트럼 데이터 압축 방법을 제공하는 것이다.

과제의 해결 수단

[0010] 상기 과제를 해결하기 위한 본 발명의 일 실시예에 의한 오로라 스펙트럼 데이터 압축 방법은,

[0011] 중앙 처리 장치(CPU)가, 오로라 스펙트럼 이미지를 복수의 열로 균등하게 분할하고 각각의 분할된 이미지를 복수의 그래픽 처리 유닛(GPU: Graphics Processing Unit)으로 전송하는 단계;

[0012] 각각의 그래픽 처리 유닛(GPU)이, 분할된 이미지의 각 픽셀의 예측값을 계산하는 단계;

[0013] 상기 각각의 그래픽 처리 유닛(GPU)이, 분할된 이미지의 각 픽셀에 대한 예측 계수를 계산하는 단계;

[0014] 상기 각각의 그래픽 처리 유닛(GPU)이, 상기 예측 계수에 기반하여 상기 각 픽셀의 예측값을 계산하는 단계;

[0015] 상기 각각의 그래픽 처리 유닛(GPU)이, 상기 분할된 이미지의 각 픽셀의 원래 값에서 상기 각 픽셀의 예측값을 감산하여 잔차를 획득하고, 상기 획득된 잔차를 상기 중앙 처리 장치(CPU)로 전송하는 단계; 및

[0016] 상기 중앙 처리 장치(CPU)가, 상기 복수의 그래픽 처리 유닛(GPU)으로부터 수신된 복수의 잔차를 인코딩하는 단계를 포함하고,

[0017] 상기 각 픽셀에 대한 예측 계수를 계산하는 단계는, 현재 픽셀 이전의 각 행의 복수의 픽셀들을 사용하여 복수의 선형 수학식을 형성하고, 상기 복수의 선형 수학식에 기반하여 상기 예측 계수를 계산한다.

[0018] 본 발명의 일 실시예에 의한 오로라 스펙트럼 데이터 압축 방법에 있어서, 상기 각 픽셀($x_{m,n}$)에 대한 예측 계수는 예측 계수 벡터($\begin{bmatrix} \alpha_0 & \alpha_1 & \cdots & \alpha_N \end{bmatrix}^T$)로부터 획득되고, 상기 예측 계수 벡터는 수학식 5에 기반하여 계산되며,

[0019] [수학식 5]

[0020]
$$\begin{bmatrix} x_{0,n-N-M+1} & x_{0,n-N-M+2} & \cdots & x_{0,n-M} \\ x_{0,n-N-M+2} & x_{0,n-N-M+3} & \cdots & x_{0,n-M+1} \\ \vdots & \vdots & \cdots & \vdots \\ x_{0,n-N} & x_{0,n-N+1} & \cdots & x_{0,n-1} \\ \vdots & \vdots & \cdots & \vdots \\ x_{m-1,n-N-M+1} & x_{m-1,n-N-M+2} & \cdots & x_{m-1,n-M} \\ x_{m-1,n-N-M+2} & x_{m-1,n-N-M+3} & \cdots & x_{m-1,n-M+1} \\ \vdots & \vdots & \cdots & \vdots \\ x_{m-1,n-N} & x_{m-1,n-N+1} & \cdots & x_{m-1,n-1} \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_N \end{bmatrix} = \begin{bmatrix} x_{0,n-M+1} \\ x_{0,n-M+2} \\ \vdots \\ x_{0,n} \\ \vdots \\ x_{m-1,n-M+1} \\ x_{m-1,n-M+2} \\ \vdots \\ x_{m-1,n} \end{bmatrix},$$

[0021] $x_{i,j}\ (i = 0, 1, \cdots, m-1,\ j = n-N, n-N+1, \cdots, n)$는 i번째 행과 j번째 열의 픽셀이고, N은 예측 차수로서 예측 계수들을 계산하는데 사용된 이전의 열 수를 나타내며, M은 $x_{m,n}$ 이전의 행들 각각으로 구성된 수학식들의 수일 수 있다.

[0022] 또한, 본 발명의 일 실시예에 의한 오로라 스펙트럼 데이터 압축 방법에 있어서, 상기 잔차를 인코딩하는 단계

는, 잔차 발생 횟수가 소정 범위 내에 있는 잔차만 인코딩하고, 상기 잔차 발생 횟수가 상기 소정 범위를 벗어나는 잔차는 인코딩하지 않고 그대로 출력하는 단계를 포함할 수 있다.

[0023] 또한, 본 발명의 일 실시예에 의한 오로라 스펙트럼 데이터 압축 방법에 있어서, 상기 수학식 5에서 좌변의 첫 번째 행렬을 C로 표시하고 예측 계수 벡터 $\begin{bmatrix} \alpha_0 & \alpha_1 & \cdots & \alpha_N \end{bmatrix}^T$ 를 $\vec{\alpha}$로 표시하며 우변의 벡터를 $\vec{b}$로 표시하면, 상기 수학식 5는 $C\vec{\alpha} = \vec{b}$ 로 표시되며, 상기 예측 계수 벡터($\vec{\alpha}$)는 수학식 2에 기반하여 계산되고,

[0024] [수학식 2]

[0025] $$\vec{\alpha} = (C^T C)^{-1} C^T \vec{b}$$

[0026] 상기 행렬 C를 $C(x_{m,n})$ 으로 표시할 때, 상기 수학식 2에서 $C^T C$ 는 $C^T(x_{m,n})C(x_{m,n})$ 로 표시되며, $C^T(x_{m,n})C(x_{m,n})$ 는 수학식 6에 기반하여 계산되고,

[0027] [수학식 6]

[0028] $$C^T(x_{m,n})\,C(x_{m,n}) = C^T(x_{m-1,n})\,C(x_{m-1,n}) + C^T{}_{m-1,n} C_{m-1,n}$$

[0029] $C_{m-1,n}$ 은

[0030] $$C_{m-1,n} = \begin{vmatrix} x_{m-1,n-M-(N-1)} & \cdots & x_{m-1,n-M} \\ \vdots & & \vdots \\ x_{m-1,n-N} & \cdots & x_{m-1,n-1} \end{vmatrix}$$ 일 수 있다.

[0031] 또한, 본 발명의 일 실시예에 의한 오로라 스펙트럼 데이터 압축 방법에 있어서, $C^T C$의 역행렬인 $(C^T C)^{-1}$ 은 가우스 조단 소거법(Gaussian Jordan Elimination Method)을 이용하여 계산될 수 있다.

발명의 효과

[0032] 본 발명의 일 실시예에 의한 오로라 스펙트럼 데이터 압축 방법에 의하면, 오로라 스펙트럼 데이터를 압축하는데 소요되는 시간을 최소화할 수 있다.

도면의 간단한 설명

[0033] 도 1은 중산 스테이션에서 캡처된 2개의 오로라 스펙트럼 이미지들을 도시한 도면으로, 각각의 오로라 스펙트럼 이미지는 1024개의 스펙트럼 밴드 및 1024개의 공간 샘플링 포인트를 가지고 있고, 우측의 컬러 바(color bar)는 각 컬러의 값을 나타내는데 사용됨.

도 2는 임의의 픽셀 $x_{m,n}$ 의 예측에 사용되는 오로라 이미지의 작은 부분을 도시한 도면.

도 3은 3개의 영역 A_1, A_2 및 A_3가 최소 제곱법을 사용하여 예측될 수 없다는 것을 나타내는 도면.

도 4는 오로라 이미지의 행을 사용하여 다수의 수학식들을 생성하는 방법을 설명하는 예를 도시한 도면.

도 5는 오리지널 온라인 DPCM 방법을 사용하는 경우 $x_{m,n}$ 의 예측 계수들을 계산하는 데 사용될 영역인

$A_1(x_{m,n})$ 및 개선된 온라인 DPCM 방법을 사용하는 경우 사용될 영역인 $A_1(x_{m,n})+A_3(x_{m,n})$을 도시한 도면.

도 6의 상부는 2014년 10월 17일, 17:41:20에 캡처된 오로라 스펙트럼 이미지의 히스토그램을 도시한 도면이고 하부는 음영 부분의 확대도를 도시한 도면으로서, 9000 근처에서 최대 픽셀 값임을 알 수 있고, 대부분의 픽셀들은 500 이내에 있음을 알 수 있으며, 대다수의 픽셀 값들은 음영 부분에 나타나지 않으며, 최대 발생 횟수는 단지 4임을 알 수 있음.

도 7의 상부는 도 6에 도시된 오로라 스펙트럼 이미지의 잔차 이미지의 히스토그램을 도시한 도면이고, 하부의 좌측은 음영부분1(shade1)의 확대도를 도시한 도면으로서, 대부분의 잔차들은 0 주위에 존재함을 알 수 있음. 하부의 우측은 음영부분2(shade2)의 확대도를 도시한 도면으로서 최대 잔차는 30000보다 크지만, 대부분의 잔차 값들은 shade2에 나타나지 않으며, 나머자 잔차들은 단지 한번 나타난다는 것을 알 수 있음.

도 8은 3개의 매개 변수들 N, M 및 T에 따른 비트레이트의 변동을 나타낸 도면으로서, 비트레이트는 각 날짜의 평균 비트레이트이고, 각 서브 도면의 5개의 날짜들은 2014 오로라 스펙트럼 데이터 세트에서 랜덤하게 선택됨.

도 9는 커널 함수의 예시적인 스레드 구조를 도시한 도면.

도 10은 행렬 C^T 및 C의 곱을 위한 분해 방법의 예를 도시한 도면.

도 11은 일련의 행렬, $C^T(x_{1,n})C(x_{1,n})$, $C^T(x_{2,n})C(x_{2,n})$, ..., $C^T(x_{m,n})C(x_{m,n})$을 계산하기 위한 방법을 도시한 도면으로서, $C_{i,n}$(i=0, 1, ..., m-1)은 열 n에 있는 픽셀들을 예측하는 경우 오로라 이미지의 행 i를 사용하여 형성된 행렬이고, 심볼 "$\oplus$"은 가산을 나타냄.

도 12의 상부는 병렬로 배열의 구간 합을 계산하는 아이디어의 예시를 도시한 도면이고, 하부는 CUDA 코드들을 도시한 도면으로서, 심볼 "$\oplus$"은 가산을 나타냄.

도 13은 스레드 블록에서 모든 워프들의 마지막 요소의 배열을 구간 합하는 병렬 구현예를 도시한 도면으로서, 각 워프의 구간 합은 도 12의 방법을 사용하여 계산되었음. 도 12를 도 13과 결합하여, 스레드 그리드 내의 각 스레드 블록의 구간 합이 획득됨.

도 14는 $C^TCBlockPrefixKernel$ 을 위한 스레드 구조를 도시한 도면.

도 15는 행렬 $C^T_{i,n}$ 및 $C_{i,n}$의 곱을 위한 CUDA 커널 함수를 도시한 도면.

도 16은 행렬 C^TC의 역행렬 계산을 위한 CUDA 커널 함수를 도시한 도면.

도 17은 9일 동안의 온라인 오로라 스펙트럼 데이터에 대한 DPCM 알고리즘 및 몇몇 다른 무손실 압축 알고리즘의 평균 비트레이트를 도시한 도면으로서, 평균은 9일 동안의 각 알고리즘의 평균 비트레이트를 나타냄.

도 18은 4개 스트림을 사용한 CUDA 애플리케이션의 예시적인 타임라인을 도시한 도면.

도 19는 오로라 이미지를 4개 스트림으로 분할하기 위한 방법을 도시한 도면.

도 20은 멀티-스트림 기법을 사용하는 온라인 DPCM 알고리즘의 CUDA 코드를 도시한 도면.

도 21은 2개 스트림, 4개 스트림, 및 디폴트 스트림을 이용하는 온라인 DPCM 알고리즘의 병렬 구현 간의 속도 비교를 도시한 도면.

도 22는 온라인 DPCM 알고리즘의 멀티-GPU 구현을 위한 코어 CUDA 코드들을 도시한 도면.

도 23은 2 GPU, 4 GPU, 및 단지 하나의 GPU를 이용한 온라인 DPCM 알고리즘의 병렬 구현 간의 속도 비교를 도시한 도면.

도 24는 본 발명의 일 실시예에 의한 오로라 스펙트럼 데이터 압축 방법의 흐름도.

도 25는 본 발명의 일 실시예에 의한 오로라 스펙트럼 데이터 압축 방법이 실행될 수 있는 장치를 도시한 도면.

발명을 실시하기 위한 구체적인 내용

[0034] 본 발명의 목적, 특정한 장점들 및 신규한 특징들은 첨부된 도면들과 연관되어지는 이하의 상세한 설명과 바람직한 실시예들로부터 더욱 명백해질 것이다.

[0035] 이에 앞서 본 명세서 및 청구범위에 사용된 용어나 단어는 통상적이고 사전적인 의미로 해석되어서는 아니되며, 발명자가 그 자신의 발명을 가장 최선의 방법으로 설명하기 위해 용어의 개념을 적절하게 정의할 수 있는 원칙에 입각하여 본 발명의 기술적 사상에 부합되는 의미와 개념으로 해석되어야 한다.

[0036] 본 명세서에서 각 도면의 구성요소들에 참조번호를 부가함에 있어서, 동일한 구성 요소들에 한해서는 비록 다른 도면상에 표시되더라도 가능한 한 동일한 번호를 가지도록 하고 있음에 유의하여야 한다.

[0037] 또한, "제1", "제2", "일면", "타면" 등의 용어는, 하나의 구성요소를 다른 구성요소로부터 구별하기 위해 사용되는 것으로, 구성요소가 상기 용어들에 의해 제한되는 것은 아니다.

[0038] 이하, 본 발명을 설명함에 있어, 본 발명의 요지를 불필요하게 흐릴 수 있는 관련된 공지 기술에 대한 상세한 설명은 생략한다.

[0039] 이하, 첨부된 도면을 참조하여 본 발명의 바람직한 실시형태를 상세히 설명하기로 한다.

[0040] 오로라의 연구 가치는 매우 높지만, 오로라 스펙트럼 데이터의 데이터 용량은 매우 커서 저장 및 전송에 큰 어려움이 있다. 이 문제를 완화하기 위해, 오로라 스펙트럼 데이터의 압축은 필수 불가결하다. 본 발명의 일 실시예에 의한 오로라 스펙트럼 데이터 압축 방법은 스펙트럼 데이터의 무손실 압축을 위한 예측 기반 온라인 차동 펄스 코드 변조(DPCM) 방법을 기반으로 하며, 이를 병렬 CUDA(Compute Unified Device Architecture)로 구현한다. 본 발명의 일 실시예 의한 오로라 스펙트럼 데이터 압축 방법은 온라인 DPCM 방법의 압축 성능을 향상시키기 위해 두 가지를 개선하였다. 하나는 예측 계수의 계산에 관한 것이고, 다른 하나는 잔차의 인코딩에 관한 것이다. CUDA 구현에서는 중복 데이터 액세스 및 계산을 피하기 위해 행렬 곱셈을 위한 분해 방법을 제안했다. 또한 CUDA 구현은 다중 스트림 기술 및 다중 그래픽 처리 유닛(GPU: Graphics Processing Unit) 기술로 각각 최적화된다. 마지막으로, 오로라 스펙트럼 이미지의 평균 압축 시간은 약 0.06 초에 이르며, 이는 15초 오로라 스펙트럼 데이터 획득 시간 간격보다 훨씬 짧으며 전송 및 기타 후속 작업에 많은 시간을 절약할 수 있다.

[0041] 본 발명의 일 실시예에 의한 오로라 스펙트럼 데이터 압축 방법에서 테스트 데이터 세트는 2014년 남극의 중산 스테이션에서 수집된 오로라 스펙트럼 데이터이다. 카메라의 시간 해상도는 15초이다. 즉, 오로라 스펙트럼 이미지는 15초마다 생성된다. 각 오로라 스펙트럼 이미지는 FITS(Flexible Image Transport System) 형식으로 저장되며 크기는 1024×1024이다. 각 픽셀은 16 비트 부호없는 정수로 저장된다. 도 1a는 2014년 7월 15일 13:29:40의 중산 스테이션에서 캡처된 오로라 스펙트럼 이미지이며 이름은 20140715_132940이다. 도 1b는 2014년 9월 23일 14:43:20에 캡처 된 오로라 스펙트럼 이미지이며 이름은 20140923_144320이다. 다음 내용에서 언급된 모든 오로라 스펙트럼 이미지는 이 이름 지정 방법을 채택한다. 오로라 스펙트럼 이미지는 15초마다 생성되므로 압축 시간은 15초 이내 이어야 한다. 이 체계는 온라인 DPCM(Differential Pulse Code Modulation) 방법을 사용하여 오로라 스펙트럼 데이터를 압축하고 CUDA(Compute Unified Device Architecture)를 사용하여 GPU(Graphics Processing Unit)에서 알고리즘을 가속화한다.

[0042] 온라인 DPCM 알고리즘의 속도를 높이기 위하여 본 발명에서는 GPU의 강력한 컴퓨팅 기능을 활용한다. 일반적으로 GPU는 그래픽 렌더링에만 책임이 있으며, 대부분의 처리는 중앙 처리 장치(CPU)에 의해 수행된다. 비디오 게임 시장과 군사 장면 시뮬레이션의 요구에 따라 GPU 성능이 빠르게 향상되었으며 GPU는 범용 GPU(GPGPU)로 개발되었다. 또한 범용 병렬 컴퓨팅 플랫폼 및 프로그래밍 모델인 CUDA의 개발을 통해 프로그래머가 고급 프로그래밍 언어 C로 프로그래밍할 수 있다.

[0043] GPU는 금융, 석유, 천문학 분야에서 중요한 역할을 한다. 신호 처리, 이미지 처리, 비디오 압축 등 GPU는 높은 산술 강도, 높은 병렬 처리 작업에 특화되어 있다. 온라인 DPCM 오로라 스펙트럼 데이터 압축 체계에서 각 픽셀의 예측 값 계산은 독립적이므로 GPU를 사용하여 병렬로 계산할 수 있다. 더욱이, 각 픽셀에 대한 예측 계수는 최소 제곱법(LSM)을 사용하여 계산되고, LSM의 주요 동작은 행렬 곱셈 및 행렬 역변환이며, 둘 다 GPU 가속에 적합하다. 따라서 온라인 DPCM 알고리즘을 GPU를 사용하여 병렬로 가속화하는 것이 합리적이고 적합하다.

[0045] **오로라 스펙트럼 데이터 압축을 위한 향상된 온라인 DPCM 방법**

[0046] 온라인 DPCM 방법 개요

[0047] 우선, 도 25에 도시된 본 발명의 일 실시예에 의한 오로라 스펙트럼 데이터 압축 방법이 실행되는 장치에 대해 설명하기로 한다.

[0048] 도 25에 도시된 본 발명의 일 실시예에 의한 오로라 스펙트럼 데이터 압축 방법이 실행되는 장치는, 오로라 스펙트럼 이미지를 복수의 열로 균등하게 분할하고 각각의 분할된 이미지를 복수의 그래픽 처리 유닛(GPU: Graphics Processing Unit)(2504_1 내지 2504_n)으로 전송하는 중앙 처리 장치(CPU)(2500), 상기 중앙 처리 장치(CPU)(2500)로부터 수신되는 분할된 이미지의 각 픽셀에 대한 예측 계수를 계산하고, 상기 예측 계수에 기반하여 상기 각 픽셀의 예측값을 계산하며, 상기 분할된 이미지의 각 픽셀의 원래 값에서 상기 각 픽셀의 예측값을 감산하여 잔차를 획득하고, 상기 획득된 잔차를 상기 중앙 처리 장치(CPU)로 전송하는 복수의 그래픽 처리 유닛(GPU: Graphics Processing Unit)(2504_1 내지 2504_n)을 포함한다.

[0049] 상기 중앙 처리 장치(CPU)(2500)는 데이터를 저장하기 위한 메모리(2502)를 포함하고, 상기 복수의 그래픽 처리 유닛(GPU)(2504_1 내지 2504_n) 각각은 데이터를 저장하기 위한 메모리(2506_1 내지 2506_n)를 포함한다.

[0050] 도 24를 참조하면, 본 발명의 일 실시예에 의한 오로라 스펙트럼 데이터 압축 방법의 주요 단계는 다음과 같다.

[0051] 단계 S2400에서, 중앙 처리 장치(CPU)(2500)가, 오로라 스펙트럼 이미지를 복수의 열(row)로 균등하게 분할하고 각각의 분할된 이미지를 제1 내지 제n 그래픽 처리 유닛(GPU: Graphics Processing Unit)(2504_1 내지 2504_n)으로 전송한다.

[0052] 단계 S2402에서, 제1 내지 제n 그래픽 처리 유닛(GPU)(2504_1 내지 2504_n) 각각은 분할된 이미지의 각 픽셀에 대한 예측 계수를 계산한다.

[0053] 단계 S2404에서, 제1 내지 제n 그래픽 처리 유닛(GPU)(2504_1 내지 2504_n) 각각은 상기 계산된 예측 계수에 기반하여 상기 각 픽셀의 예측값을 계산한다.

[0054] 단계 S2406에서, 제1 내지 제n 그래픽 처리 유닛(GPU)(2504_1 내지 2504_n) 각각은 상기 분할된 이미지의 각 픽셀의 원래 값에서 상기 각 픽셀의 예측값을 감산하여 잔차를 획득하고, 상기 획득된 잔차를 상기 중앙 처리 장치(CPU)(2500)로 전송한다.

[0055] 상기 중앙 처리 장치(CPU)(2500)는 상기 제1 내지 제n 그래픽 처리 유닛(GPU)(2504_1 내지 2504_n)으로부터 수신된 잔차들을 인코딩한다.

[0056] 상기 각 픽셀에 대한 예측 계수를 계산하는 단계 S2402는, 현재 픽셀 이전의 각 행의 복수의 픽셀들을 사용하여 복수의 선형 수학식을 형성하고, 상기 복수의 선형 수학식에 기반하여 상기 예측 계수를 계산한다.

[0057] 이하, 첨부된 도면들을 참조하여 본 발명의 일 실시예에 의한 오로라 스펙트럼 데이터 압축 방법에 대해 상세히 설명하기로 한다.

[0058] 상기 예측 계수는 최소 제곱법(LSM)을 사용하여 계산된다. 도 2는 픽셀 $x_{m,n}$과 픽셀을 예측하는 데 사용되는 영역을 보여주며, 수학식 1은 $x_{m,n}$에 대한 예측 계수를 계산하기 위해 만들어진 수학식들의 선형 시스템이다. N은 예측 차수이며 예측 계수들을 계산하는 데 사용된 이전의 열 수를 정의한다. $x_{i,j}\ (i = 0, 1, \cdots, m-1,\ j = n-N, n-N+1, \cdots, n)$는 i번째 행과 j번째 열의 픽셀이며 $\begin{bmatrix} \alpha_0 & \alpha_1 & \cdots & \alpha_N \end{bmatrix}^T$는 예측 계수 벡터이다. 수학식들의 선형 시스템은 영역 $A_1(x_{m,n})$의 모든 픽셀을 포함하지만 $x_{m,n}$ 자체는 포함하지 않음을 알 수 있다. 그러므로, 디코더는 또한 $A_1(x_{m,n})$의 픽셀을 사용하여 $x_{m,n}$의 예측 계수를 계산할 수 있고, 예측 계수를 디코더에 전송할 필요가 없다. 이것이 온라인 압축의 아이디어이다.

수학식 1

[0059]
$$\begin{bmatrix} x_{0,n-N} & x_{0,n-N+1} & \cdots & x_{0,n-1} \\ x_{1,n-N} & x_{1,n-N+1} & \cdots & x_{1,n-1} \\ \vdots & \vdots & \cdots & \vdots \\ x_{m-1,n-N} & x_{m-1,n-N+1} & \cdots & x_{m-1,n-1} \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_N \end{bmatrix} = \begin{bmatrix} x_{0,n} \\ x_{1,n} \\ \vdots \\ x_{m-1,n} \end{bmatrix}$$

[0060] 그런 다음 수학식 1에서 첫 번째 행렬을 C로 표시하고 예측 계수 벡터 $\begin{bmatrix} \alpha_0 & \alpha_1 & \cdots & \alpha_N \end{bmatrix}^T$ 를 $\vec{\alpha}$ 로 표시하며, 수학식 우변의 벡터를 $\vec{b}$ 로 표시하면, 수학식 1의 압축 형식, 즉 $C\vec{\alpha} = \vec{b}$ 를 얻으며, 마지막으로, 수학식 2를 사용하여 예측 계수 벡터 $\vec{\alpha}$ 가 계산될 수 있다.

수학식 2

[0061]
$$\vec{\alpha} = (C^T C)^{-1} C^T \vec{b}$$

[0062] 수학식 2로부터, 예측 계수들의 계산은 하나의 행렬 곱셈, 하나의 행렬 역변환 및 두 개의 행렬-벡터 곱셈을 요구한다$((C^T C)^{-1}(C^T \vec{b}))$. 예측 계수의 계산은 실제로 DPCM에서 가장 많은 시간이 소요되는 부분이다. 픽셀의 예측값의 계산은 훨씬 간단하다. 예를 들어, 도 2의 픽셀 $x_{m,n}$ 의 예측값은 영역 $A_2(x_{m,n})$ 의 픽셀의 선형 조합이다. 이를 설명하기 위하여 수학식 3이 사용된다. $\hat{x}_{m,n}$ 은 $x_{m,n}$ 의 예측값이다.

수학식 3

[0063]
$$\hat{x}_{m,n} = \sum_{i=0}^{N-1} \alpha_{i+1} x_{m,n-N+i}$$

[0064] 그러나 도 3에 도시된 3개의 영역(A_1, A_2, A_3)과 같은 오로라 이미지의 일부 에지 픽셀에 대한 예측 값은 수학식 3을 사용하여 계산할 수 없다. 왜냐하면, 이러한 에지 픽셀에 대한 예측 계수를 계산하기 위해 수학식 1과 같은 선형의 수학식 시스템을 구축할 수 없기 때문이다. 도 3에서 H는 오로라 이미지의 높이이고, W는 너비, A_1은 행 0, A_2는 열 0이다. 행 0 앞에는 행이 없고 열 0 앞에는 열이 없으므로, A_1 및 A_2는 LSM을 사용하여 예측될 수 없다. 열 1 앞에 열이 하나만 있기 때문에 A_3은 열 1이다. 다시 말해, 선형 수학식 시스템을 구축하면 등호 왼쪽에 단 하나의 항목만 있으며 LSM을 사용하여 A_3을 예측하지 않을 것이다. 수학식 4는 각 영역에 대한 예측 방법을 나타낸다. $\hat{x}_{m,n}$ 는 $x_{m,n}$ 의 예측값이다. A_1 및 A_3에서, 각 픽셀의 예측값은 픽셀 $x_{0,0}$을 제외하고 열 방향으로 인접한 픽셀의 값이다. A_2에서, 각각의 픽셀의 예측값은 행 방향으로 인접한 픽셀의 값이다.

수학식 4

$$A_1 : \hat{x}_{0,0} = x_{0,0}, \hat{x}_{0,n} = x_{0,n-1}, n \geq 1$$
$$A_2 : \hat{x}_{m,0} = x_{m-1,0}, m \geq 1$$
$$A_3 : \hat{x}_{m,1} = x_{m,0}, m \geq 1$$

[0065]

[0066] 잔차는 원래 값과 예측값의 차이이다. 인코딩 방법은 범위(range) 코더이다. 온라인 DPCM 방법에서, 인코딩되고 디코더로 전송되어야 하는 유일한 데이터는 잔차이다.

[0068] 기존 온라인 DPCM 방법 개선

[0069] 실험을 통해 온라인 DPCM 알고리즘의 압축 성능이 두 가지 개선으로 향상될 수 있음을 발견했다. 하나는 예측 계수를 계산할 때 선형 수학식 시스템의 설정을 개선하는 것이고 다른 하나는 잔차 인코딩을 개선하는 것이다.

[0071] 선형 수학식 시스템의 설정 개선

[0072] 수학식 1로부터, 픽셀 $x_{m,n}$ 의 각각의 이전 행으로 단 하나의 수학식만 구축되지만, 실험을 통해 이전의 행들 각각으로 여러 개의 수학식들을 형성할 때 압축 성능이 향상될 수 있음이 발견되었다. 오로라 이미지의 각 행으로 여러 수학식들을 형성하는 방법을 설명하기 위해 도 4에 도시된 예가 사용된다. $x_{m,n}$은 예측 계수들을 계산할 픽셀이다. M은 $x_{m,n}$ 의 이전 행들 각각으로 구성된 수학식들의 수이다. M=3, 예측 차수 N=4 및 열 수 n=8이라고 가정하면 행 i(i = 0, 1, ..., m-1)로 형성된 M개의 수학식들은 (*)이다(도 4 참조). 길이가 N+1인 창을 행에서 연속으로 슬라이딩할 때 이러한 수학식들이 획득될 수 있다.

[0073] 픽셀 $x_{m,n}$ 앞의 각 행으로 구성된 수학식들을 결합하면 $x_{m,n}$ 의 예측 계수들을 계산하는 데 사용되는 수학식들의 완전한 선형 시스템을 얻을 수 있으며 수학식들의 선형 시스템은 수학식 5에 표시된다. 빨간색 점선 상자의 M개의 수학식들은 오로라 이미지의 한 행으로 만들어진다. 각각의 빨간색 점선 박스는 오로라 이미지의 행의 N+M 개의 상이한 요소들을 포함하는 것을 알 수 있는데, 다시 말해서, 행 m 앞의 각 행의 연속적인 N+M 개의 요소들이 $x_{m,n}$ 의 예측 계수들을 계산하는데 사용될 것임을 알 수 있다. 도 5는 이를 보다 명확하게 설명하는 데 사용된다. 도 2와 비교하여, 개선된 방법은 더 많은 영역을 필요로 하는데, 즉 열 수가 M-1인 $A_3(x_{m,n})$을 필요로 하고, $A_1(x_{m,n})$ 및 $A_3(x_{m,n})$의 총 열 수는 N+M이다. 수학식 5는 수학식 1에 대한 개선이므로, 수학식 5는 $C\vec{\alpha} = \vec{b}$ 로서 표시되고, 또한, 이하의 내용에서 언급될 C, $\vec{\alpha}$ 및 $\vec{b}$는 수학식 5에서의 C, $\vec{\alpha}$ 및 $\vec{b}$이다. 행렬 C의 열 수는 예측 차수 N이고, 행렬 C의 행들의 수는 M×m이며 행 번호 m에 따라 다르다.

수학식 5

[0074]
$$\begin{bmatrix} x_{0,n-N-M+1} & x_{0,n-N-M+2} & \cdots & x_{0,n-M} \\ x_{0,n-N-M+2} & x_{0,n-N-M+3} & \cdots & x_{0,n-M+1} \\ \vdots & \vdots & \cdots & \vdots \\ x_{0,n-N} & x_{0,n-N+1} & \cdots & x_{0,n-1} \\ \vdots & \vdots & \cdots & \vdots \\ x_{m-1,n-N-M+1} & x_{m-1,n-N-M+2} & \cdots & x_{m-1,n-M} \\ x_{m-1,n-N-M+2} & x_{m-1,n-N-M+3} & \cdots & x_{m-1,n-M+1} \\ \vdots & \vdots & \cdots & \vdots \\ x_{m-1,n-N} & x_{m-1,n-N+1} & \cdots & x_{m-1,n-1} \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_N \end{bmatrix} = \begin{bmatrix} x_{0,n-M+1} \\ x_{0,n-M+2} \\ \vdots \\ x_{0,n} \\ \vdots \\ x_{m-1,n-M+1} \\ x_{m-1,n-M+2} \\ \vdots \\ x_{m-1,n} \end{bmatrix}$$

[0076] 잔차의 인코딩 개선

[0077] 본 발명에서 압축할 데이터는 2014년 중산 스테이션에서 캡처한 오로라 스펙트럼 데이터이지만 압축에 부정적인 영향을 줄 수 있는 단점이 있음을 발견했다. 각 오로라 스펙트럼 이미지에는 다른 픽셀보다 훨씬 큰 특이치들이 있지만 도 6과 같이 몇 번만 나타난다. 이러한 특이치들은 예측하기 어렵기 때문에 압축에 해로운 영향을 끼치는 매우 큰 잔차를 발생시킨다. 이 문제를 처리하기 위해 양수 및 음수 잔차에 대한 임계값을 각각 설정하는 이중 임계값 방법을 제안한다. 두 임계 값 사이의 잔차만 범위 코더를 사용하여 인코딩되고 다른 잔차는 압축 파일에 그대로 저장된다.

[0078] 도 7은 이중 임계값 방법의 세부 사항을 보여준다. T는 잔차 발생 횟수의 임계 값이다. T_-는 왼쪽에서 오른쪽으로 검색할 때 발생 횟수가 T보다 크거나 같은 첫 번째 잔차이고, T_+는 오른쪽에서 왼쪽으로 검색할 때 발생 횟수가 T보다 크거나 같은 첫 번째 잔차이다. T_-와 T_+ 사이의 잔차만 범위 코더를 사용하여 인코딩되며 나머지 잔차는 압축 파일에 그대로 저장된다.

[0080] 파라미터 N, M 및 T의 최적화

[0081] 매개 변수 N은 현재 픽셀을 예측하는 데 사용할 열 수를 결정하고 M은 오로라 스펙트럼 이미지의 각 행으로 만들 수 있는 수학식들의 수를 정의한다. 이 두 매개 변수는 예측 정확도에 중요한 영향을 미친다. 임계값(T)은 인코딩에 영향을 줄 두 개의 임계값(T_- 및 T_+)을 결정한다. 최상의 압축 성능을 얻으려면 실험을 통해 이러한 매개 변수 N, M 및 T를 최적화해야 한다. 실험 데이터 세트는 2014 오로라 스펙트럼 데이터 세트이다. 오로라가 매일 나타나지 않기 때문에 2014 오로라 스펙트럼 데이터 세트에는 20일 이상의 데이터가 있으며 매일 수백 개의 이미지가 있다. 계산량을 줄이기 위해 매일의 이미지에서 100개의 샘플을 골라서 실험에 사용하였다.

[0082] 도 8a는 예측 차수 N에 따른 비트 전송률의 변화를 보여준다. 비트 전송률은 매일의 데이터에서 고르게 선택된 100 샘플의 평균 비트 전송률이며 bpp(픽셀 당 비트 수)로 측정된다. 가로 세로 좌표 9, 10, 11 및 12의 4개의 값은 예비 실험에 의해 결정된다. 각 라인의 5점 별표 기호는 매일의 평균 비트 전송률을 최소화하는 최적의 N을 나타내며, 5점 별표 근처의 숫자는 최소 비트 전송률이다. 4일 동안 최적의 N은 11이고 1일 동안 최적의 N은 10이라는 것을 알 수 있다. 매일 평균 비트 전송률을 최소화하는 최종 N을 결정하기 위해 5일의 비트 전송률을 평균했으며, 그 결과는 도 8a에서 위에서 아래로 세 번째 줄이다. N = 11은 5일의 평균 비트 전송률을 최소화하므로 2014 오로라 스펙트럼 데이터 압축의 예측 차수로서 N = 11이 사용된다.

[0083] 도 8b는 도 8a와 비슷하지만 오로라 스펙트럼 이미지의 각 행으로 몇 개의 수학식이 형성되는지를 정의하는 M에 따른 비트 전송률의 변화를 보여준다. M에 대해 1, 2, ..., 10의 10개의 값이 있다. 매일의 비트 전송률을 최소화하는 최적의 M은 다르지만 5일의 평균 비트 전송률을 최소화하는 M은 7이라는 것을 알 수 있다. 2014 오로라 스펙트럼 데이터의 압축에는 M=7이 사용된다.

[0084] M=1 일 때 매일의 비트 전송률은 도 8b의 각 줄에서 첫 번째 다이아몬드 노드 근처에 표시된다. 최소 비트 전송

률과 M=1 일 때의 비트 전송률의 차이는 0.1 bpp 이상에서 0.2 bpp 이상이며, M=1 일 때 5일의 평균 비트 전송률은 5일의 최소 평균 비트 전송률보다 약 0.17bpp 더 크다. 결과적으로, 오로라 스펙트럼 이미지의 각 행에 여러 개의 수학식을 형성하는 개선이 압축에 유리하다는 결론을 내릴 수 있다.

[0085] 도 8c는 2개의 인코딩 임계값 T_- 및 T_+를 결정하는 T에 의한 비트 전송률의 변화를 도시한다. 가로 세로 좌표 0, 5, 6, ..., 15에는 12개의 값이 있다. T=0은 임계값이 없으며 모든 잔차가 범위 코더를 사용하여 인코딩될 것이라는 것을 나타낸다. 매일의 비트 전송률을 최소화하는 최적의 T는 다르지만, T가 13일 때 5일의 평균 비트 전송률의 최소값이 달성된다는 것을 알 수 있다. 따라서 2014 오로라 스펙트럼 데이터의 압축에 T=13이 선택될 것이다. 또한, T=0에서 매일의 비트 전송률은 각 라인의 첫 번째 다이아몬드 노드 근처에 표시된다. T=0에서의 비트 전송률이 각 라인의 최소 비트 전송률보다 크고, T=0 일 때 5일의 평균 비트 전송률이 5일의 최소 평균 비트 전송률보다 약 0.06bpp 크다는 것을 알 수 있다. 따라서, 인코딩을 위한 이중 임계값 방법은 효과적이다.

[0086] 또한, 매일의 평균 비트 전송률은 약 5.8bpp에서 약 7.2bpp로 크게 변할 수 있음을 알 수 있다. 이는 오로라가 강도를 포함하여 끊임없이 변화하기 때문이다. 도 1a 및 도 1b는 매우 다르다는 것을 알 수 있다.

[0088] 향상된 온라인 DPCM 알고리즘을 사용하는 예측 계수 계산의 GPU 구현

[0089] 온라인 DPCM 알고리즘에서 가장 시간이 많이 걸리는 부분은 예측 계수의 계산이다. 이는 각 픽셀의 예측 계수가 선형 수학식 시스템을 풀면서 얻어지기 때문이다. 또한, 이것은 온라인 방법이므로, 디코더는 또한 각 픽셀에 대한 예측 계수들을 계산할 필요가 있다. 따라서 예측 계수들의 계산을 가속화해야 하며 본 발명에서는 GPU의 강력한 컴퓨팅 기능을 활용한다.

[0090] 수학식 2로부터, 예측 계수들의 계산에서 주요 동작은 행렬 곱셈 및 행렬 역변환이다. 따라서 이 섹션의 나머지 부분에서는 이 두 단계에 중점을 두지만 그 전에 다음 설명을 용이하게 하기 위하여 GPU 프로그래밍의 몇 가지 기본 개념을 소개한다.

[0092] GPU 프로그래밍의 일부 기본 개념

[0093] GPU는 계산 집약적인 작업에 특화되어 있다. GPU의 프로그래밍 모델은 CUDA이고 프로그래밍 언어는 CUDA C이다. C 함수와 비슷하지만 GPU에서 실행되는 커널 함수들은 대규모 계산을 처리하기 위해 병렬의 수십만 개의 CUDA 스레드에 의해 레이블이 지정된다. NVIDIA(GPU 회사)는 이 아키텍처를 SIMT(Single Instruction Multiple Thread)로 표시한다.

[0094] 커널 함수에 할당된 모든 스레드는 2계층 그리드 구조로 구성된다. 다수의 스레드가 스레드 블록을 형성하고 다수의 스레드 블록이 스레드 그리드를 형성한다. 스레드 그리드의 모든 스레드 블록 크기는 동일하다. 스레드 블록과 스레드 그리드는 모두 1차원, 2차원 또는 3차원일 수 있다. 도 9는 스레드 구조를 설명하는 예를 보여준다. 스레드 그리드의 크기는 3 x 2이고 각 스레드 블록의 크기는 4×3이다. 괄호 안의 순서 쌍은 스레드 또는 스레드 블록의 인덱스이다.

[0095] CUDA 프로그램의 일반적인 처리 흐름은 다음과 같다. (1) 처리할 데이터를 호스트(CPU 메모리)에서 디바이스(GPU 메모리)로 복사한다. (2) 커널 함수를 시작하여 GPU에서 데이터를 처리한다. (3) 결과를 디바이스에서 호스트로 복사한다. 또한 CUDA 프로그램의 성능을 공유하는 데 매우 중요한 메모리 유형이 있다. 공유 메모리는 온칩 메모리이며 액세스 대기 시간은 몇 클럭 사이클에 불과하고, 이는 수백 클록 전역 메모리 사이클의 액세스 지연보다 훨씬 적다. 또한, 공유 메모리는 스레드 블록의 모든 스레드에 대한 통신 수단을 제공한다.

[0097] 분해 방법을 이용한 행렬 C^T와 C의 곱셈의 CUDA 구현

[0098] 행렬 곱셈은 CUDA 프로그래밍에서 일반적인 작업이며 매우 잘 최적화되었다. 기본 선형 대수 서브프로그램(cuBLAS)은 CUDA를 위한 선형 대수 가속 라이브러리로 행렬 곱셈 뿐만 아니라 행렬 일괄 처리의 행렬 곱셈도 처리할 수 있다. 수학식 2에서 행렬 C^T와 C의 곱은 각 픽셀을 예측하는 데 필요하며, 각 오로라 스펙트럼 이미지에는 1024×1024 픽셀이 있다. 그러나 행렬 C^T와 C의 곱셈은 cuBLAS 라이브러리를 사용하여 일괄 처리할 수 없다. 행렬 C의 크기가 고정되어 있지 않기 때문이다. 그것은 수학식 5에 도시된 바와 같이 예측될 픽셀의 행 번

호에 의해 결정되며, cuBLAS 라이브러리는 동일한 차원을 갖는 일련의 행렬만을 처리할 수 있다. 또한, 각 픽셀에 대한 행렬 C^T 및 C의 곱셈이 cuBLAS 행렬 곱셈 함수를 사용하여 개별적으로 수행되는 경우, 시작 오버 헤드가 너무 무겁다. 따라서 행렬 C^T와 C의 곱셈에 cuBLAS 라이브러리를 채택하지 않았으므로 본 발명에서는 상황에 적합한 체계를 제안한다.

[0099] 수학식 5로부터, m번째 행 및 n번째 열에서 픽셀 $x_{m,n}$을 예측할 때, 행 m 이전의 모든 행은 행렬 C에 관여할 것이다. 설명의 편의를 위해 행렬 C를 $C(x_{m,n})$으로 레이블을 지정한다. 유사하게, 행 m+1 이전의 모든 행은 $x_{m+1,n}$을 예측할 때 행렬 $C(x_{m+1,n})$에 포함된다. 또한 행렬 $C(x_{m+1,n})$의 첫 번째 m×M 행은 정확히 행렬 $C(x_{m,n})$이므로 행렬 $C^T(x_{m,n})$($C(x_{m,n})$의 전치 행렬) 및 $C(x_{m,n})$의 곱셈 및 행렬 $C^T(x_{m+1,n})$와 $C(x_{m+1,n})$의 곱셈은 직접적이고 개별적으로 수행되며 많은 반복 데이터 액세스 인스턴스와 계산이 도입되어, 계산 효율에 심각한 영향을 미칠 것이다. 따라서, 본 발명에서는 이 문제를 처리하기 위하여 행렬 C^T와 C의 곱셈을 위한 분해 방법을 제안한다.

[0100] 도 10은 행렬 C^T와 C의 곱셈에 대한 분해 방법을 보여주는 예를 보여준다. 오로라 스펙트럼 이미지의 왼쪽 위 모서리의 작은 부분이 도 10의 오른쪽 위 모서리에 표시된다. 단순화를 위해 N=3이고 M=2이며, 세 개의 픽셀 $x_{1,8}$, $x_{2,8}$, $x_{3,8}$을 예측해야 하는 것으로 가정한다. $C(x_{1,8})$, $C(x_{2,8})$ 및 $C(x_{3,8})$은 각각 3개의 픽셀 $x_{1,8}$, $x_{2,8}$ 및 $x_{3,8}$에 대한 행렬 C를 나타내고, $C^T(x_{1,8})$, $C^T(x_{2,8})$ 및 $C^T(x_{3,8})$은 각각 $C(x_{1,8})$, $C(x_{2,8})$ 및 $C(x_{3,8})$에 대한 전치 행렬이다. 행렬 $C(x_{1,8})$의 두 행은 함께 그룹화되어 $C_{0,8}$로 표시된다. $C_{0,8}$의 첨자 "0"은 두 개의 행이 오로라 이미지의 행 0으로 형성됨을 나타내고 첨자 "8"은 예측할 픽셀이 열 8에 있음을 나타낸다. 마찬가지로 $C(x_{2,8})$은 2개의 서브 행렬 $C_{0,8}$ 및 $C_{1,8}$로 분할되고, $C(x_{3,8})$은 3개의 서브 행렬 $C_{0,8}$, $C_{1,8}$ 및 $C_{2,8}$로 분할된다. 따라서, 행렬 $C^T(x_{1,8})$과 $C(x_{1,8})$의 곱은 $C^T{}_{0,8}C_{0,8}$이고, $C^T(x_{2,8})$과 $C(x_{2,8})$의 곱은 $C^T{}_{0,8}C_{0,8} + C^T{}_{1,8}C_{1,8}$이며 $C^T(x_{1,8})C(x_{1,8}) + C^T{}_{1,8}C_{1,8}$로 간주될 수 있다. $C^T(x_{3,8})$과 $C(x_{3,8})$의 곱은 $C^T{}_{0,8}C_{0,8} + C^T{}_{1,8}C_{1,8} + C^T{}_{2,8}C_{2,8}$이며 $C^T(x_{2,8})C(x_{2,8}) + C^T{}_{2,8}C_{2,8}$로도 간주될 수 있다. 행렬 $C^T(x_{m,n})$과 $C(x_{m,n})$의 곱셈은 많은 작은 행렬의 가산으로 변환되고, $C^T(x_{m,n})C(x_{m,n})$은 수학식 6에 기재된 바와 같이 $C^T(x_{m-1,n})C(x_{m-1,n})$과 $C^T{}_{m-1,n}C_{m-1,n}$의 합이다. 이런 식으로, 모든 행렬 $C^T(x_{i,n})C(x_{i,n})$(i=1, 2, ..., m)은 행렬의 배열 $C^T{}_{0,n}C_{0,n}, C^T{}_{1,n}C_{1,n}, \ldots, C^T{}_{m-1,n}C_{m-1,n}$의 구간 합(prefix sum)을 통해 획득될 수 있다.

수학식 6

[0101] $$C^T(x_{m,n})C(x_{m,n}) = C^T(x_{m-1,n})C(x_{m-1,n}) + C^T{}_{m-1,n}C_{m-1,n}$$

[0102] $C_{m-1,n}$은

[0103] $$C_{m-1,n} = \begin{vmatrix} x_{m-1,n-M-(N-1)} & \cdots & x_{m-1,n-M} \\ \vdots & & \vdots \\ x_{m-1,n-N} & \cdots & x_{m-1,n-1} \end{vmatrix}$$ 이다.

[0104] 이 방법의 아이디어는 도 11에 직관적으로 제시되어 있다.

[0105] 도 11에 도입된 방법을 가속화하기 위해, 본 발명에서는 도 12에 도시된 병렬 방식이 이용된다. 두 행렬의 합산의 목적은 요소들을 동일한 위치에 가산하는 것이다. 유사하게, 일련의 행렬의 구간 합의 목적은 행렬에서 동일한 위치에 있는 모든 요소로 구성된 배열의 구간 합을 구하는 것이다. 따라서 도 12에 소개된 방법은 실제로 배열의 구간 합을 위한 병렬 방법이다. 도 12의 상단 부분은 8개의 요소가 있는 배열의 구간 합을 계산하는 방법을 보여주는 예이다. i~j (i<j)는 i번째 요소 내지 j번째 요소의 합을 나타낸다. 8개 요소의 구간 합이 3번의 반복을 통해 완료되었음을 알 수 있다.

[0106] 도 12의 하단 부분은 워프(warp)에서 요소들의 구간 합을 위한 CUDA 코드를 보여준다. __shfl_up_sync()는 CUDA 셔플(shuffle) 명령으로, 동일한 워프의 스레드들이 추가 메모리를 소비하지 않고 직접 데이터를 교환할 수 있도록 하며, 공유 메모리보다 대기 시간이 짧다. 이것이 __shfl_up_sync()를 사용하는 이유이다. __shfl_up_sync()의 최대 동작 범위는 전체 워프이지만, 실제로 워프는 몇 개의 스레드 그룹으로 균등하게 나눌 수 있으며 __shfl_up_sync()는 각 스레드 그룹에서 개별적으로 실행된다. 도 12의 CUDA 코드의 너비는 각 스레드 그룹의 크기이며 정확히 32인 warpSize로 설정된다. __shfl_up_sync (마스크, 값, i, 너비)의 함수는 스레드 lane_id - i의 값을 가져오는 것이다. CUDA 코드의 본문은 for 루프임을 알 수 있다. i번째 반복에서, 첫 번째 i 스레드를 제외한 모든 스레드는 __shfl_up_sync()를 호출하여 스레드 lane_id-i가 보유한 value의 값을 얻은 다음, 획득한 값을 자체 value에 추가한다. 5번 반복한 후, 32개의 스레드가 있는 워프의 구간 합이 완료된다. 그러나 이것은 워프에서 구간 합만을 달성한다; 1023 행렬의 구간 합을 계산하기 위하여 (각 오로라 이미지에는 1024개의 행이 있음) 각 스레드 블록 내에서 구간 합을 계산하고 마지막으로 전체 스레드 그리드에서 구간 합을 계산해야 한다.

[0107] 도 13은 스레드 블록 구간 합의 후반, 즉 스레드 블록에 있는 모든 워프의 마지막 요소의 구간 합과 결과 배열의 각 요소를 해당 워프에 가산하는 것을 보여준다. 도 13의 상단 부분은 일반적인 아이디어를 나타내고, 하단 부분은 CUDA 코드를 나타낸다. 코드는 세 부분으로 나눌 수 있다. 첫 번째 부분은 도 12의 작업 후에 각 워프의 마지막 요소를 저장하는 value를 공유 메모리 sdata에 복사하는 것이다. 두 번째 부분은 sdata에서 요소들의 구간 합을 하는 것이다. 각 스레드 블록에는 nwarps 워프들이 있으며 __shfl_up_sync()의 마지막 매개 변수 width는 nwarps로 설정된다. 마지막 부분은 sdata의 각 요소를 해당 워프에 가산하는 것이다. 워프 0 전에 아무런 워프도 없기 때문에, 워프 0에 0이 가산된다.

[0108] 도 12와 도 13의 코드들을 결합하여 각 스레드 블록 내에서 구간 합을 계산하기 위한 CUDA 커널 함수가 획득되고, 커널은 CTCBlockPrefixKernel로 표시된다. CTCBlockPrefixKernel의 스레드 구조는 도 14에 나와 있다. 스레드 그리드의 각 페이지는 오로라 이미지의 열을 처리하며 총 WIDTH-2 페이지가 있다. 이는 처음 두 열이 도 3에 도시된 바와 같이 미리 예측되기 때문이다. 각 스레드 블록에 nthreads 스레드가 있다고 가정한다. 오로라 이미지의 첫 번째 행도 사전에 예측되므로 오로라 이미지의 각 열에 HEIGHT-1C^TC 행렬이 있으며 스레드 그리드의 y차원은 (HEIGHT-1+block.x-1)/block.x이다. 각 C^TC 행렬은 N×N(N은 예측 차수) 요소를 가지므로 구간 합을 위해 N×N 배열이 존재하며, 스레드 그리드의 x차원은 N×N이다.

[0109] 도 14의 스레드 블록들의 열에서 구간 합을 계산하는 방식은 도 13에 설명된 방법과 유사하다. 먼저 각 스레드 블록의 마지막 요소들의 구간 합을 계산한 다음, 결과 배열의 각 요소를 해당 스레드 블록에 가산한다. 이 절차 후에 행렬 $C^T(x_{m,n})C(x_{m,n})\ (m = 1, 2, \ldots, 1023, n = 2, 3, \ldots, 1023)$ 가 획득될 수 있다(오로라 이미지의 첫 번째 행과 처음 두 열은 도 3에 도시된 바와 같이, 미리 예측된다).

[0110] 앞의 내용은 $C^T(x_{m,n}) \times C(x_{m,n})$ 을 많은 작은 행렬 $C^T{}_{i,n}C_{i,n}$ (i=0, 1, ..., m-1)의 합으로 분해하는 방법에 대해 설명했지만, 행렬 $C^T{}_{i,n}$ 및 $C_{i,n}$ 의 곱셈은 여전히 문제로 남아있다. 도 10에서 $C_{i,n}$의 크기는 M×N

이며, M은 오로라 이미지의 각 행으로 만들 수학식들의 수를 결정하고 N은 예측 차수이다. 따라서 행렬 $C^T_{i,n}C_{i,n}$ 의 차원은 N×N이다. 또한 상기에 설명된 바와 같이 2014 오로라 데이터 세트에 대한 최적의 N은 11이므로 각 $C^T_{i,n}C_{i,n}$ 에는 11×11=121개의 요소들이 있다. 이것은 최근 GPU 카드의 각 스레드 블록에서 지원하는 최대 스레드 수(1024)보다 훨씬 적다. 결과적으로, 스레드 블록은 $C^T_{i,n}$ 및 $C_{i,n}$의 곱셈에 충분하며 각 스레드는 곱셈 행렬 $C^T_{i,n}C_{i,n}$ 의 요소를 담당한다. 물론, 행렬 $C^T_{i,n}C_{i,n}$ 의 차원이 각 스레드 블록에 의해 지원되는 최대 스레드 수보다 큰 경우, 곱셈 행렬 $C^T_{i,n}C_{i,n}$ 은 작은 서브 행렬로 나눌 수 있고 각 스레드 블록은 하나의 서브 행렬을 처리한다.

[0111] 도 15는 행렬 $C^T_{i,n}$ 및 $C_{i,n}$의 곱셈의 CUDA 구현을 제공한다. 스레드 블록의 크기는 N×N이며, 이는 곱셈 행렬 $C^T_{i,n}C_{i,n}$ 의 크기와 동일하다. 각 스레드는 $C^T_{i,n}C_{i,n}$ 의 한 요소를 담당한다, 즉, 각 스레드는 $C^T_{i,n}$ 행과 $C_{i,n}$ 열을 곱하는 역할을 한다. 스레드 그리드의 크기는 (HEIGHT-1)×(WIDTH-2)이며, 이는 오로라 이미지의 첫 번째 행과 첫 번째 두 열이 미리 예측되기 때문이다. 오로라 이미지의 처음 몇 열은 M개의 행과 N개의 열을 가진 행렬 $C_{i,n}$을 설정할 수 없으므로, cols와 rows는 각각 $C_{i,n}$의 실제 너비와 높이를 나타내는 데 사용된다. col(오로라 이미지의 현재 열 색인)이 N보다 작거나 같으면, cols는 col이고 rows는 1이며, 그렇지 않으면 cols는 N이고 rows는 col-N+1이며, 이는 오로라 이미지의 각 행으로 형성될 수 있는 수학식들의 최대 수이다. 하지만, rows가 M보다 큰 경우 rows는 M으로 설정된다. $C_{i,n}$ 의 모든 요소는 도 10에 도시된 바와 같이, 오로라 이미지의 행 i의 연속 세그먼트에서 온 것이므로, 세그먼트는 글로벌 메모리 d_data의 액세스를 감소시키기 위하여 그리고 공유 메모리의 낮은 레이턴시를 이용하기 위해 공유 메모리 sdata에 복사된다. 이런 식으로 행렬 $C_{i,n}$을 미리 준비할 필요가 없고 호스트에서 장치로 복사할 필요가 없으며, 호스트에서 장치로 전송해야 하는 유일한 데이터는 오로라 데이터이므로 많은 시간이 절약된다.

[0112] 지금까지 행렬 곱셈의 소개가 완료되었지만, 이 분해 아이디어는 또한, 수학식 2에서 C^T와 $\vec{b}$의 곱셈에 적용될 수 있다. 이것은 도 10에 도시된 바와 같이 $C(x_{i-1,n})$ 이 $C(x_{i,n})$의 프리픽스이고, $\vec{b}(x_{i-1,n})$ ($C(x_{i-1,n})$ 에 대응하는 우변 벡터) 또한 $\vec{b}(x_{i,n})$ 의 프리픽스이기 때문이다.

[0114] 가우스 조단 소거법을 이용한 행렬 C^TC의 역변환의 CUDA 구현

[0115] 수학식 2에 도시된 바와 같이, 행렬 C^TC의 역변환은 오로라 이미지의 픽셀의 예측 계수를 계산할 때 가장 중요한 절차 중 하나이다. 본 발명에서 사용된 C^TC의 역변환 방법은 가우스 조단 소거법(Gaussian Jordan Elimination)이며, 이와 관련하여 CUDA를 사용하여 병렬로 가속하는 방법에 대해 설명한다. 가우스 조단 소거법의 주요 단계는 동일한 차원의 항등 행렬을 C^TC의 오른쪽으로 확장한 다음 제거를 통해 C^TC의 각 열을 단위 벡터로 순차적으로 변경하는 것이다. 행렬 C^TC가 최종적으로 항등 행렬이 되면, 오른쪽 절반 행렬은 역행렬이다. 즉, $[C^TC \vdots E] \overset{elimination}{\rightarrow} [E \vdots (C^TC)^{-1}]$, E는 N-차원 단위 행렬이다. 또한 반올림 오차를 제어하고 0으로 나누기를 피하기 위해 제거 전에 주 요소를 선택해야 하며, 편의상 주 요소를 선택하기 위해 채택한 범위는 열이다.

[0116] 가우스 조단 소거법이 선택된 이유는 행렬 C^TC의 차원이 N이고, 2014 오로라 스펙트럼 데이터 세트에 대한 최적의 N이 11이므로, C^TC는 중소형 행렬이기 때문이다. C^TC의 역행렬을 계산하기 위해 수반 행렬 방법을 사용하는

경우 N×N 대수 보조 인자와 C^TC의 결정자를 계산해야 한다. 이는 많은 양의 계산이다. LU 분해(행렬을 단위 하부 삼각 행렬과 상부 삼각 행렬로 분해하는) 또는 QR 분해(행렬을 직교 행렬과 상부 삼각 행렬로 분해하는)와 같은 분해 방법을 사용하는 경우, 먼저 C^TC를 여러 하위 행렬로 분해한 다음 각 하위 행렬의 역행렬을 계산하고 마지막으로 이러한 역행렬의 곱을 계산할 필요가 있다. 분해 방법의 각 단계에는 적어도 하나의 커널 함수 호출이 필요하다. 따라서 분해 방법은 매우 복잡하고 처리 시간은 대형 행렬의 경우 가우스 조단 소거법의 처리 시간보다 짧을 수 있지만 중소형 행렬 C^TC의 경우 커널 함수의 시작 오버 헤드가 분해 방법의 이점을 상쇄할 수 있다.

[0117] C^TC의 역행렬 계산을 위한 가우스 조단 방법의 CUDA 구현은 도 16에 나와 있다. 각 스레드 블록은 행렬 C^TC의 역행렬을 계산하고 스레드 블록의 크기는 (N×2)×N으로서, 증대된 행렬 $invC^TC$의 크기와 동일하다; cols는 행렬 C^TC 실제 차원이다. 먼저, 행렬 C^TC가 $invC^TC$의 왼쪽 절반에 복사되고, $invC^TC$의 오른쪽 절반은 단위 행렬로 설정된다. 그런 다음 for 루프를 반복할 때마다 $invC^TC$의 한 열이 가우스 조단 소거법을 통해 단위 벡터로 바뀐다. C^TC의 실제 열 수는 cols이기 때문에 for 루프에 cols 반복이 있다. 각 반복의 첫 번째 단계는 열의 주요 요소를 검색하는 것이다. 열에 N=11개의 요소만 있고 k번째 반복에서 주요 요소의 검색 범위는 행 k에서 마지막 행까지이므로, 주요 요소의 검색 범위가 줄어들기 때문에, 열의 주요 요소를 순차적으로 검색하기 위하여 스레드 0이 지정된다. 주요 요소가 0이면 0으로 나누기 오류가 발생할 것이고, 행렬 C^TC는 특이 행렬로 간주된다. 공유된 메모리 kk는 주요 요소를 저장하는데 사용되고, pivot는 주요 요소의 행 인덱스이다. 행 인덱스입니다. 열의 주요 요소가 계산되면, 행 pivot이 정규화되고 행 k와 교환되어야 한다. 행 pivot은 제거하는 동안 여러 번 사용될 것이기 때문에, 전역 메모리의 액세스를 줄이기 위하여, 행 pivot을 공유 메모리 srow에 복사한다. 이런 식으로 srow에 대해 정규화가 수행되고, 행 pivot과 행 k는 srow의 도움으로 교환될 수 있다. 마지막 단계는 제거이다. 행 k를 제외하고, 행렬 $invC^TC$의 각 행에서 행 pivot의 특정 비율을 빼서, 이 행의 k번째 요소를 0으로 만든다. 실제로 이 비율은 행 pivot의 k번째 요소에 대한 각 행의 k번째 요소의 비율이다. 행 pivot은 정규화되고 k번째 요소가 1로 바뀌었으므로, 상기 비율은 각 행의 k번째 요소가 된다. 열 k가 제거 동안 여러 번 액세스된다는 것을 알 수 있고, 따라서 전역 메모리의 심각한 액세스 대기 시간을 피하기 위해 그것은 공유 메모리 scol에 복사된다. cols 반복 후, $invC^TC$의 왼쪽 절반이 단위 행렬로 바뀌고, 오른쪽 절반은 정확하게 C^TC의 역행렬이다.

[0119] 실험 결과

[0120] 실험을 통해 CUDA 구현의 성능을 검증하였다. 테스트 데이터 세트는 2014년 중산 스테이션에서 캡처한 오로라 스펙트럼 데이터 세트이다. 직렬 프로그램을 실행하는 CPU는 주요 주파수가 3.4GHz인 Intel Xeon E3-1240 v3 CPU이다. GPU는 3584 CUDA 코어와 12GB GDDR5X 메모리를 가진 NVIDIA TITAN X GPU이며 메모리 대역폭은 480GB/s이다.

[0122] 다른 무손실 압축 알고리즘과 비교하여 개선된 온라인 DPCM 알고리즘의 압축 성능

[0123] 개선된 온라인 DPCM 알고리즘의 세부 사항은 앞서 소개되었으며, 3개의 매개 변수 N, M 및 T는 각각 최적의 값, 즉 N=11, M=7 및 T=13으로 설정되어 있다. 도 17은 온라인 DPCM 알고리즘과 몇 가지 다른 무손실 압축 알고리즘의 성능을 나타낸 것이다. Bzip2와 PPMD(Dmitry에 의해 구현된 부분 매칭에 의한 예측 알고리즘)은 7-Zip 소프트웨어에서 구현된다. Bzip2는 버로우-휠러(Burrows-Wheeler) 변환(BWT) 무손실 데이터 압축 알고리즘에 기반한다. PPMD는 정보 상속(Information Inheritance)을 사용한 PPM(부분 매칭에 의한 예측) (PPMII) 알고리즘을 약간 변형한 구현이며, PPMII는 PPM 알고리즘을 수정하여 높은 계산 복잡성을 줄인 것이다. JPEG-LS 및 CALIC은 이전에 간략하게 소개되었다. 매일의 비트 전송률은 매일의 데이터에서 고르게 선택된 100개의 오로라 스펙트럼 이미지의 평균 비트 전송률이며 비트 전송률이 작을수록 압축 성능이 향상되었다. 온라인 DPCM의 비트 전송률은 9일 중 6일 동안 가장 낮으며, 이 9일의 온라인 DPCM의 평균 비트 전송률은 다른 모든 알고리즘의 비트 전송률보다 낮다. 따라서 온라인 DPCM이 도 17에 나와있는 다른 모든 알고리즘보다 성능이 우수하다는 결론을 내릴 수

있다.

[0125] 온라인 DPCM 알고리즘의 병렬 구현 성능

[0126] CUDA 프로그램의 경우, 가장 우려하는 것은 프로그램과 대응하는 직렬 프로그램 간의 속도 비교이다. 표 1은 온라인 DPCM 알고리즘의 병렬 구현 및 2개의 직렬 구현에 대한 실행 시간을 보여준다. speedup은 병렬 시간에 대한 직렬 분해 시간의 비율을 나타낸다. 각 날짜의 시간은 실제로 각 날짜의 데이터에서 고르게 선택된 100개의 오로라 스펙트럼 이미지의 평균 시간이다.

표 1

[0127]

Data \ Implementation	Serial Direct Time (s)	Serial Decomposed Time (s)	Parallel Time (s)	Speedup
20140328	869.16	3.75	0.19	19.7
20140404	872.56	3.74	0.21	17.8
20140425	872.28	3.74	0.21	17.8
20140629	872.24	3.74	0.21	17.8
20140705	872.12	3.74	0.21	17.8
20140718	869.67	3.74	0.21	17.8
20140730	870.48	3.73	0.21	17.8
20140827	872.68	3.74	0.21	17.8

[0128] "직렬 분해" 구현의 목적은 이전에 소개된 분해 아이디어를 행렬 곱셈 C^TxC 및 행렬-벡터 곱셈 $C^T \times \vec{b}$(수학식 2에 표시된)에 적용하는 것이고, "직렬 직접" 구현의 목적은 중복 계산을 고려하지 않고 상기 행렬 곱셈과 상기 행렬-벡터 곱셈을 직접 계산하는 것이다. 이러한 직렬 구현들은 모두 Intel Xeon E3-1240 v3 CPU에서 테스트되었다. 직렬 직접 구현의 실행 시간은 약 870초이며, 이는 직렬 분해 구현의 3.74초의 실행 시간보다 훨씬 길다. 그러므로, 본 발명의 분해 방법은 중복 계산을 피하고 많은 시간을 절약할 수 있는 매우 효과적인 방법이다. 병렬 구현은 TITAN X GPU에서 테스트되었다. 표 1에서 병렬 구현의 실행 시간은 약 0.21초인데, 이것은 15초 오로라 데이터 수집 시간 간격보다 훨씬 짧으므로 각 오로라 스펙트럼 이미지에 대해 다른 작업을 수행하는 데 많은 시간을 절약할 수 있다.

[0129] 표 1에 제시된 병렬 시간은 모든 커널 함수의 실행 시간과 호스트와 장치 간 데이터 전송 시간을 포함하여, 병렬 애플리케이션의 총 실행 시간이다. 또한, 표 2는 2개의 오로라 이미지들에 대한 3개의 CUDA 커널 함수들과 2개의 데이터 전송의 시간과 비율을 보여준다.

표 2

[0130]

	20140514_133241		20140923_155500	
	Time	Ratio (%)	Time	Ratio (%)
C^TCMulKernel	10.820 ms	5.71	10.759 ms	5.71
C^TCBlockPrefixKernel	11.920 ms	6.29	11.857 ms	6.29
matrixInverseKernel	137.29 ms	72.41	136.42 ms	72.36
memcpyHtoD	191.24 us	0.10	194.25 us	0.10
memcpyDtoH	161.32 us	0.09	161.32 us	0.09

[0131] C^TCMulKernel은 도 15에 표시된 커널 함수이고, C^TCBlockPrefixKernel (도 12 및 도 13에 도시됨)은 스레드 블록에서 배열의 구간 합을 계산하기 위한 커널 함수이며, matrixInverseKernel (도 16에 도시됨)은 행렬 C^TC의 역행렬을 계산하기 위한 커널 함수이다; memcpyHtoD는 호스트에서 디바이스로의 데이터 전송이며, 호스트에서 디바이스로 전송되어야 하는 유일한 데이터는 오로라 스펙트럼 데이터이다. 또한, memcpyDtoH는 디바이스에서 호스트로의 데이터 전송, 즉 디바이스에서 호스트로의 잔차의 전송이다. matrixInverseKernel이 가장 많은 시간을 소비하고, 이 세 가지 커널 함수가 소비한 총 시간이 84% 이상이며, 호스트와 디바이스 간 데이터 전송은 총 시간의 약 0.2%만 소요됨을 알 수 있다. 따라서 커널 함수를 실행하는 데는 거의 모든 시간이 걸린다고 할 수 있다.

[0132] 속도 외에도, achieved_occupancy 및 gld_throughput과 같이 CUDA 프로그램의 성능을 측정하는 다른 메트릭도 있다. 이 둘은 nvprof라는 CUDA 명령 라인 프로파일러에 의해 제공된다. achievement_occupancy는 스트리밍 멀티프로세서(SM)의 점유율을 측정하는 데 사용되며, gld_throughput은 전역 메모리 로드 처리량이다.

[0133] 표 3은 2개의 오로라 이미지들에 대한 세 가지 커널 함수 C^TCMulKernel, C^TCBlockPrefixKernel 및 matrixInverseKernel의 achievement_occupancy 및 gld_throughput을 보여준다.

표 3

	20140514_133241		20140923_155500	
	achieved_occupancy	*gld_throughput*	*achieved_occupancy*	*gld_throughput*
C^TCMulKernel	87.06%	5.43 GB/s	87.24%	4.61 GB/s
C^TCBlockPrefixKernel	94.88%	321.91 GB/s	94.88%	310.63 GB/s
matrixInverseKernel	99.63%	183.53 GB/s	99.63%	183.21 GB/s

[0134]

[0135] 이 세 가지 커널 함수들의 achieved_occupancy는 매우 높고, matrixInverseKernel의 achieved_occupancyd은 99.63%임을 알 수 있다. gld_throughput의 경우 C^TCBlockPrefixKernel의 처리량은 300GB/s 이상이고 matrixInverseKernel의 처리량은 180GB/s 이상이며, C^TCMulKernel의 처리량은 약 5GB/s이다. 이것은 C^TCMulKernel (도 15에 도시됨)에서 각 스레드 블록이 $C^T_{i,n}$ 및 $C_{i,n}$의 곱셈을 담당하고, $C_{i,n}$을 구성하는 연속적인 N+M-1 요소들(도 5에 도시됨)이 공유 메모리에 복사되기 때문이다. 결과적으로, 전역 메모리의 액세스가 크게 줄어든다.

[0137] 멀티-스트림 기법을 이용한 온라인 DPCM 알고리즘의 병렬 구현

[0138] CUDA 스트림은 호스트 코드에 의해 시작된 순서대로 디바이스에서 실행되는 일련의 CUDA 작업이다. 기본적으로 하나의 CUDA 스트림만 있으며, 모든 데이터가 이 스트림에서 처리된다. 여러 스트림을 사용하려면, 명시적으로 여러 스트림을 생성하고, 처리할 모든 데이터를 여러 개의 작은 부분으로 나누며, 각 부분을 스트림에 분배해야 한다. 다중 스트림을 사용하는 이점은 다른 스트림에서 CUDA 연산의 실행 순서가 임의적이므로, 다른 스트림에서 CUDA 연산의 실행을 겹치게 하여 전체 애플리케이션의 실행 시간을 줄일 수 있다는 것이다. 도 18은 다른 스트림에서 CUDA 작업들의 오버랩을 보다 명확하게 설명하는 예이다. 4개의 스트림이 있으며 각 스트림에는 하나의 커널 함수만 있다. HtoD는 호스트에서 디바이스로의 데이터 전송을 나타내고 DtoH는 디바이스에서 호스트로의 데이터 전송을 나타낸다. 데이터 전송 및 커널 실행의 중첩, 커널 실행의 중첩 및 다른 방향으로의 데이터 전송의 중첩의 세 가지 유형의 중첩이 있음을 알 수 있다.

[0139] 멀티-스트림 기술을 사용한 온라인 DPCM 알고리즘의 최적화를 위해 오로라 이미지를 열로 균등하게 나누고 각 부분을 스트림에 할당하는 것은 도 5에 도시된 바와 같이, 오로라 이미지의 픽셀 예측은 그 앞에 있는 모든 행들을 포함하기 때문이다. 오로라 이미지를 4개의 스트림으로 나누는 방법은 도 19에 나와 있으며, 2개의 스트림이 있는 경우 나누기 방법은 비슷하다. 도 20은 멀티-스트림 기술의 CUDA 코드를 보여준다. C^TCMulKernel, C^TCBlockPrefixKernel, matrixInverseKernel 등을 포함한 모든 CUDA 커널은 입력이 오로라 데이터이고 출력이 잔차인 하나의 커널과 같다. 오로라 데이터는 동기 함수 cudaMemcpy()를 사용하여 디바이스에 복사된다. 이 함수는 각 스트림이 그것에 할당된 데이터를 호스트에서 디바이스로 비동기적으로 비동기 함수 cudaMemcpyAsync()가 아니라, 데이터 전송이 완료될 때까지 호스트를 차단하여 각 스트림에 할당된 데이터를 복사할 수 있도록 한다. 이는 오로라 데이터가 행별로 저장되지만 열별로 각 스트림에 할당되기 때문이며, 그래서 각 스트림에 할당된 오로라 데이터가 호스트 상에 사전에 준비되어 있지 않으면 각 스트림은 cudaMemcpyAsync()를 한 번 호출하여 디바이스에 할당된 오로라 데이터를 복사할 수 없다. cudaMemcpy()를 사용하여 잔차를 복사하는 이유도 비슷하다.

[0140] 도 21은 2개의 스트림과 4개의 스트림을 사용하는 온라인 DPCM 알고리즘의 병렬 구현의 실행 시간을 보여준다. 각 날짜의 시간은 각 날짜의 데이터에서 고르게 선택된 100개의 오로라 스펙트럼 이미지의 평균 시간이다. 기본 스트림을 사용하는 속도는 표 1에 나와 있으며, 여기서 목표는 여러 스트림을 사용하는 속도와 비교하는

것이다. 이 세 가지 구현의 속도는 기본적으로 동일하다는 것을 알 수 있다. 이는 전체 시간의 84% 이상을 차지하는(표 2) 세 개의 커널 C^{T}CMulKernel, C^{T}CBlockPrefixKernel 및 matrixInverseKernel의 SM 점유율이 동일하고, 다른 커널의 SM의 점유율도 실제로 매우 높기 때문이다. 멀티-스트림 프로그램에서, 한 스트림의 커널 함수가 시작된 경우 사용 가능한 SM이 충분할 때만 다른 스트림의 커널 함수가 시작될 수 있다. 그렇지 않으면 커널 함수들이 중첩되지 않으므로 총 실행 시간을 줄이는데 기여할 수 없다.

[0142] 다중 GPU 기술을 사용한 온라인 DPCM 알고리즘의 병렬 구현

[0143] 여러 GPU로 계산하려면 프로그래머는 큰 계산 문제를 여러 개의 작은 하위 문제로 세분화하고 각 하위 문제를 GPU에 배포해야 한다. 모든 GPU가 동시에 동작하므로 총 실행 시간이 줄어든다. 여러 GPU를 사용하여 온라인 DPCM 알고리즘을 구현할 때 계산 작업을 세분화하는 방법은 여러 스트림들을 사용할 때와 동일하다. 즉, 오로라 이미지를 열로 균등하게 나누고 각 부분을 GPU에 배포한다. 그러나 모든 스트림이 동일한 GPU 메모리를 사용하는 멀티-스트림 프로그래밍과 달리 각 GPU에는 자체 메모리가 있다. 따라서 도 20과 같이 오로라 데이터는 cudaMemcpy()를 한 번 호출하여 디바이스로 전송될 수 없다. cudaMemcpyAsync()를 사용하여 각 GPU에 할당된 오로라 데이터를 각 GPU의 메모리로 전송해야 하며, 전송 전에 준비의 목적은 호스트에서 각 GPU에 대한 데이터를 미리 준비하는 것이지만, 이것은 시간이 더 걸린다. 또한 각 GPU에서 계산된 잔차는 각 GPU의 메모리에 저장되므로 cudaMemcpy() 대신 cudaMemcpyAsync()를 사용하여 호스트로 전송해야 한다. 다중 GPU 구현의 일부 핵심 CUDA 코드가 도 22에 나와 있다.

[0144] 도 22의 첫 번째 부분은 각 GPU에 대한 데이터를 준비하는 것이다. nColsPerGPU는 각 GPU에 할당된 열의 수이다. dataCols는 각 GPU가 픽셀들의 nColsPerGPU 열을 예측하는 데 필요한 열의 수이며, GPU 0을 제외하고, 이것은 nColsPerGPU+N+M-1이다. 이는 도 5에 도시된 바와 같이 각 픽셀이 예측하기 위해 N+M-1 이전의 열이 필요하기 때문이다. 하지만, GPU 0에 할당된 데이터 이전에는 데이터가 없으며, 실제 사용 가능한 데이터를 사용하여 GPU 0의 데이터의 초기 몇 열이 예측된다. cudaMallocHost()는 고정 된 호스트 메모리를 할당하는 CUDA 함수이며, h_auroraData[i]에 고정된 호스트 메모리를 할당하는 이유는 cudaMemcpyAsync()가 호스트 메모리를 고정된 메모리로 요구하기 때문이다. 두 번째 부분은 각 GPU의 오로라 데이터 및 잔차를 저장하기 위해 디바이스 메모리를 할당하고, 각 GPU에 대한 스트림을 생성하는 것이다. 먼저, 각각의 GPU에 필요한 오로라 데이터는 고정된 호스트 메모리 h_auroraData[i]로부터 각 디바이스로 비동기식으로 전송된다. 그런 다음 커널 함수가 시작되어 C^{T}CMulKernel, C^{T}CBlockPrefixKernel, matrixInverseKernel 등 모든 커널에 해당하는 잔차를 계산한다. 마지막으로, 잔차는 각 디바이스에서 고정된 호스트 메모리 h_residual로 비동기식으로 복사된다.

[0145] 2개의 GPU와 4개의 GPU를 사용하는 다중 GPU 구현 속도는 표 4에 나와 있다. 표 4는 온라인 DPCM 알고리즘의 직렬 및 멀티-GPU 구현에 대한 실행 시간을 나타낸 것이다.

표 4

[0146]

		20140514	20140705	20140827	20140916	20140930	20141017
Serial time(s)		3.74	3.74	3.74	3.74	3.74	3.74
2 GPUs	Time(s)	0.12	0.12	0.12	0.12	0.12	0.12
	Speedup	31.2	31.2	31.2	31.2	31.2	31.2
4 GPUs	Time(s)	0.06	0.06	0.07	0.07	0.07	0.07
	Speedup	62.3	62.3	53.4	53.4	53.4	53.4

[0147] 각 날짜의 시간은 각 날짜의 데이터에서 고르게 선택된 100개의 오로라 스펙트럼 이미지의 평균 시간이다. GPU 2개 사용시 실행 시간은 약 0.12초이고 GPU 수가 4개로 증가하면 실행 시간은 0.06초 내지 0.07초이며 15초 오로라 데이터 수집 시간 간격보다 훨씬 적다. 속도 향상은 각각 2 GPU 시간 및 4 GPU 시간에 대한 직렬 시간의 비율이다.

[0148] 또한 하나의 GPU, 2개의 GPU 및 4개의 GPU를 사용하는 속도 비교는 도 23에 나와 있다. 하나의 GPU를 사용하는 성능은 앞서 언급하였으며 여기서는 여러 GPU를 사용하는 것과 비교된다. GPU 수를 두 배로 늘리면 시간이 약 절반으로 줄어드는 것을 알 수 있다. GPU 수를 두 배로 늘리면 각 GPU에서 처리하는 데이터 양이 절반으로 줄어들기 때문이다. 시간이 정확히 반으로 감소되지는 않지만 항상 반보다 약간 크게 감소된다. 이는 GPU 수가 증가

함에 따라 장치 시작 오버 헤드 및 커널 시작 오버 헤드와 같은 오버 헤드도 증가하기 때문이다.

[0149] 본 발명은 오로라 스펙트럼 데이터의 무손실 압축을 위한 예측 기반 온라인 DPCM 알고리즘의 병렬 CUDA 구현에 관한 것이다. 온라인 DPCM 방법의 첫 번째 단계는 최소 제곱법을 사용하여 예측 계수를 계산한 다음 예측 계수를 사용하여 각 픽셀의 예측 값을 계산하고 원래 값에서 예측 값을 뺌으로써 잔차를 얻고, 마지막으로 잔차를 인코딩하는 것이다. 온라인 DPCM 알고리즘의 압축 성능을 향상시키기 위해 두 가지 개선 사항을 제안했다. 첫 번째는 예측 계수를 계산할 때 현재 픽셀 이전의 각 행을 사용하여 여러 수학식을 만드는 것이고, 다른 하나는 잔차의 인코딩에 이중 임계값 방법을 적용하는 것이다. 온라인 DPCM의 CUDA 구현에서는 행렬 C^T와 C의 곱셈과 행렬 C^TC의 역변환에 중점을 두었다. 중복된 데이터 액세스 및 계산을 제거하기 위해 C^T와 C의 곱셈을 많은 작은 행렬의 가산으로 분해하는 분해 방법이 제안되었다. 또한, 멀티-스트림 기술 및 멀티 GPU 기술은 CUDA 구현을 최적화하는 데 사용되었다. 마지막으로, 4개의 GPU를 사용할 때 오로라 스펙트럼 이미지의 평균 압축 시간은 약 0.06초이다.

[0150] 이상 본 발명을 구체적인 실시예를 통하여 상세하게 설명하였으나, 이는 본 발명을 구체적으로 설명하기 위한 것으로, 본 발명은 이에 한정되지 않으며, 본 발명의 기술적 사상 내에서 당 분야의 통상의 지식을 가진 자에 의해 그 변형이나 개량이 가능함은 명백하다고 할 것이다.

[0151] 본 발명의 단순한 변형 내지 변경은 모두 본 발명의 영역에 속하는 것으로, 본 발명의 구체적인 보호 범위는 첨부된 청구범위에 의하여 명확해질 것이다.

부호의 설명

[0152] 2500 : CPU

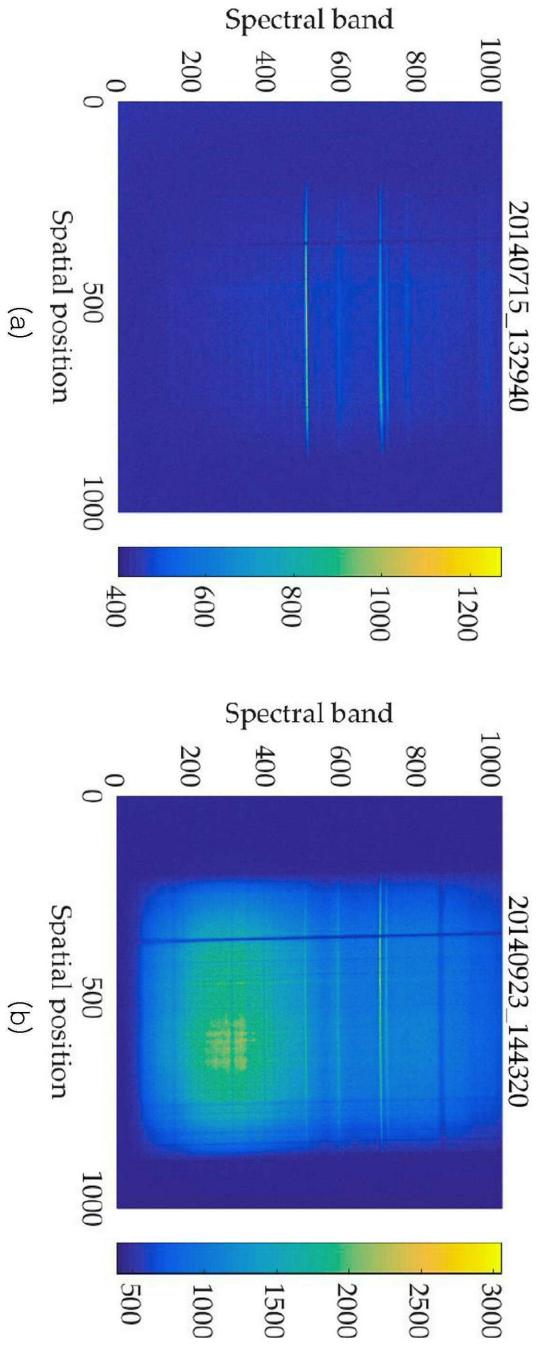
2502 : CPU 메모리

2504 : GPU

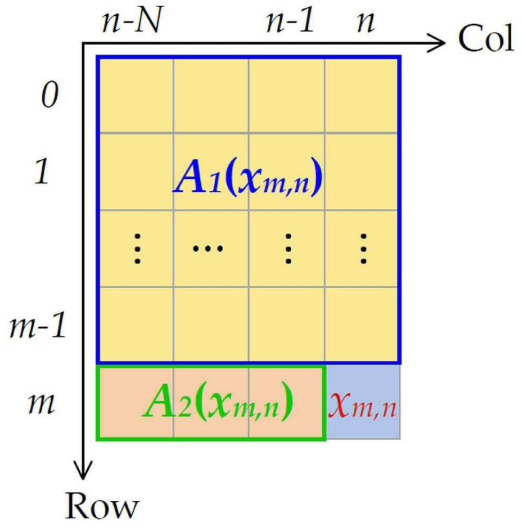
2506 : GPU 메모리

도면

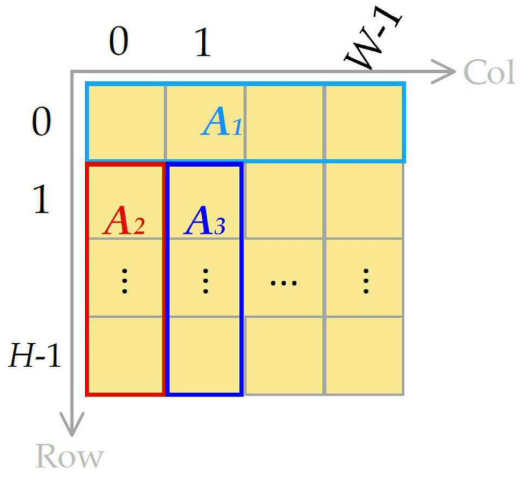
도면1


Spectral band
0
200
400
600
800
1000
Spatial position
0
500
1000
20140715_132940
400
600
800
1000
1200
(a)
Spectral band
0
200
400
600
800
1000
Spatial position
0
500
1000
20140923_144320
500
1000
1500
2000
2500
3000
(b)

도면2



도면3



도면4

//M defines how many equations to build with each row of an aurora image

Conditions: $n = 8$, $N = 4$, $M = 3$

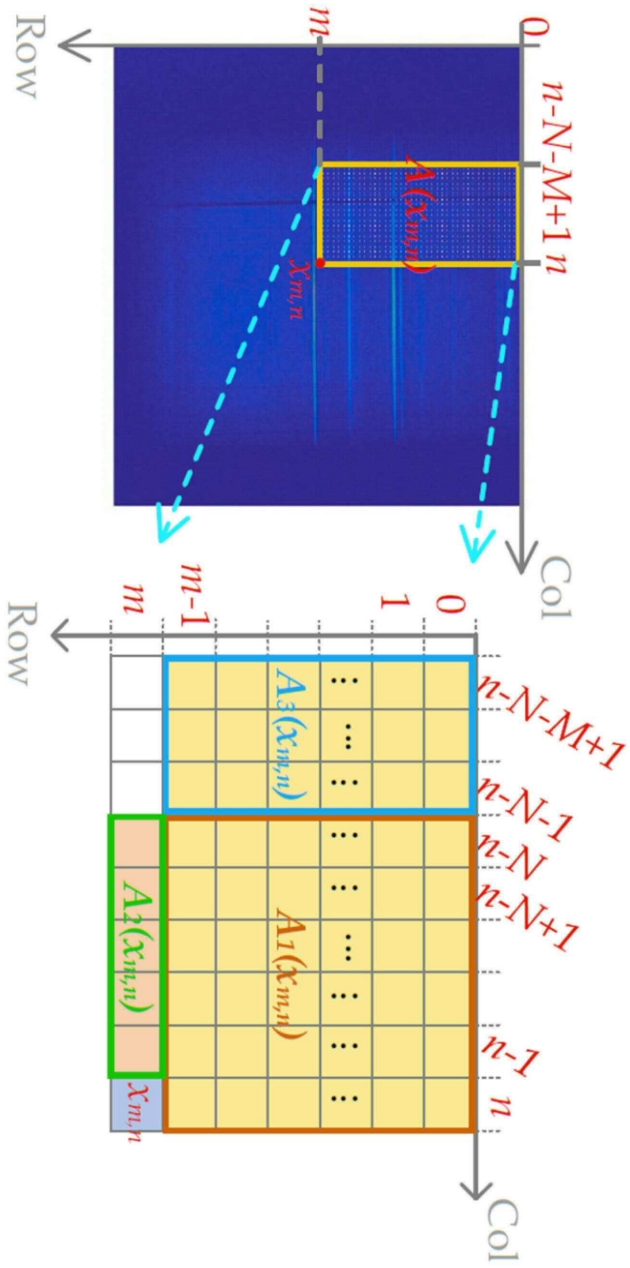
A part of row i ($i = 0, 1, \cdots, m - 1$) of an aurora image:

$$x_{i,0} \; x_{i,1} \; x_{i,2} \; x_{i,3} \; x_{i,4} \; x_{i,5} \; x_{i,6} \; x_{i,7} \; x_{i,8}$$

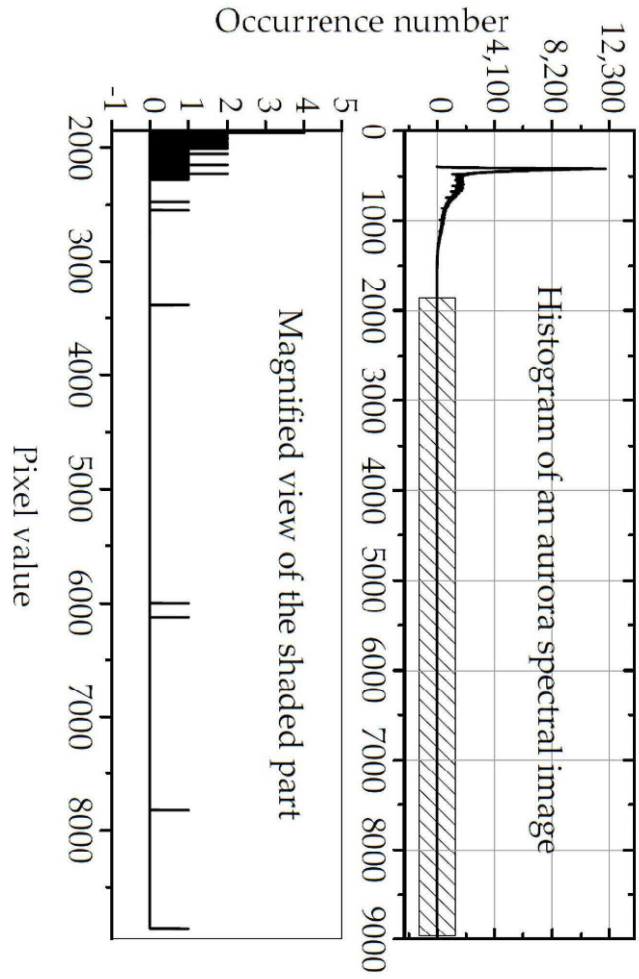
The M equations built with row i:

$$\begin{bmatrix} x_{i,2} & x_{i,3} & x_{i,4} & x_{i,5} \\ x_{i,3} & x_{i,4} & x_{i,5} & x_{i,6} \\ x_{i,4} & x_{i,5} & x_{i,6} & x_{i,7} \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \end{bmatrix} = \begin{bmatrix} x_{i,6} \\ x_{i,7} \\ x_{i,8} \end{bmatrix} \quad (*)$$

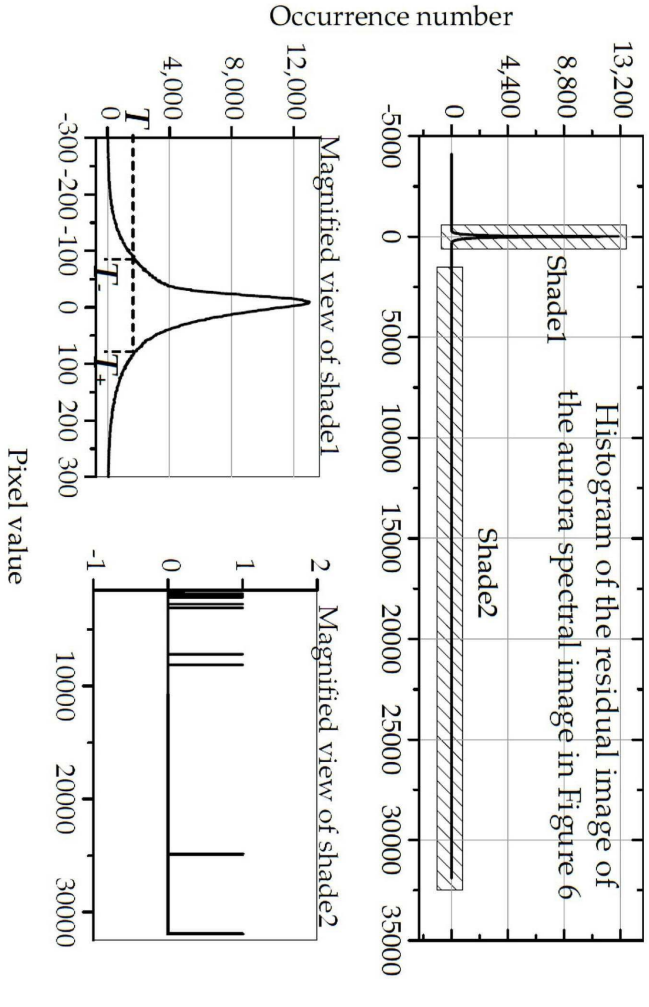
도면5


Row
Col
m
0
n-N-M+1
n
A(x_{m,n})
x_{m,n}
m
m-1
1
0
n-N-M+1
n-N-1
n-N
n-N+1
n-1
n
A3(x_{m,n})
A2(x_{m,n})
A1(x_{m,n})
x_{m,n}

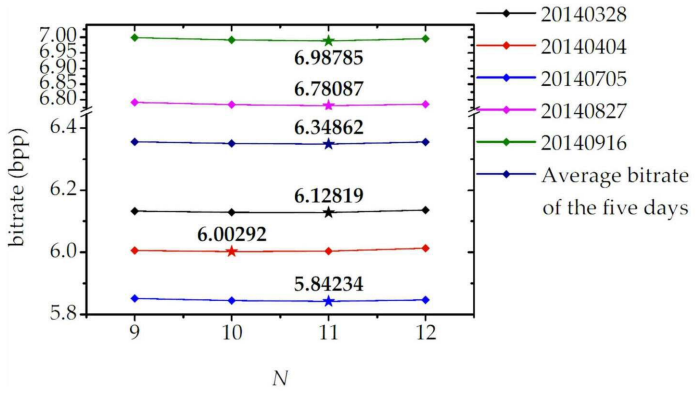
도면6


Occurrence number
12,300
8,200
4,100
0
0 1000 2000 3000 4000 5000 6000 7000 8000 9000
Histogram of an aurora spectral image
5
4
3
2
1
0
-1
2000 3000 4000 5000 6000 7000 8000
Magnified view of the shaded part
Pixel value

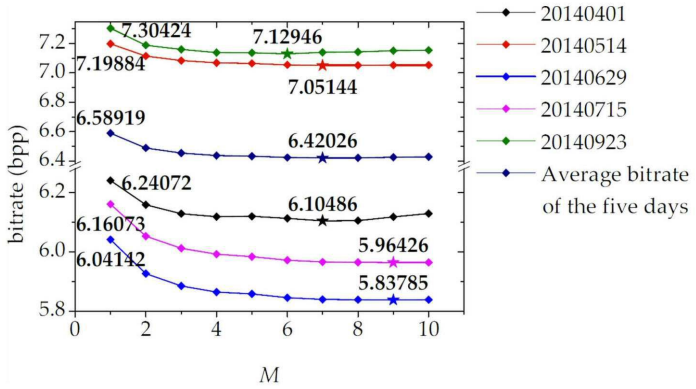
도면7


Occurrence number
Histogram of the residual image of the aurora spectral image in Figure 6
Shade1
Shade2
Magnified view of shade1
Magnified view of shade2
Pixel value

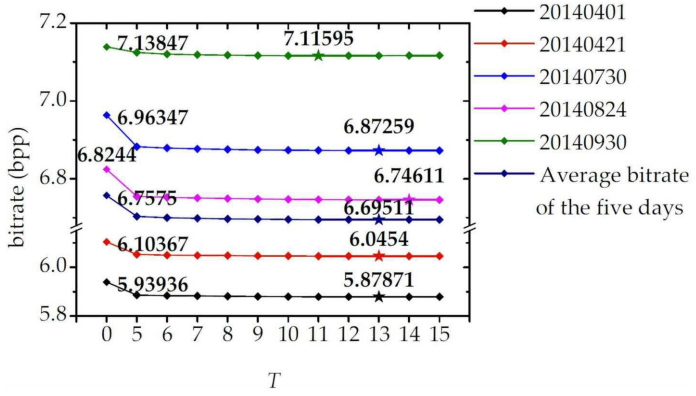
도면8


20140328
20140404
20140705
20140827
20140916
Average bitrate
of the five days
6.98785
6.78087
6.34862
6.12819
6.00292
5.84234
bitrate (bpp)
N

(a)

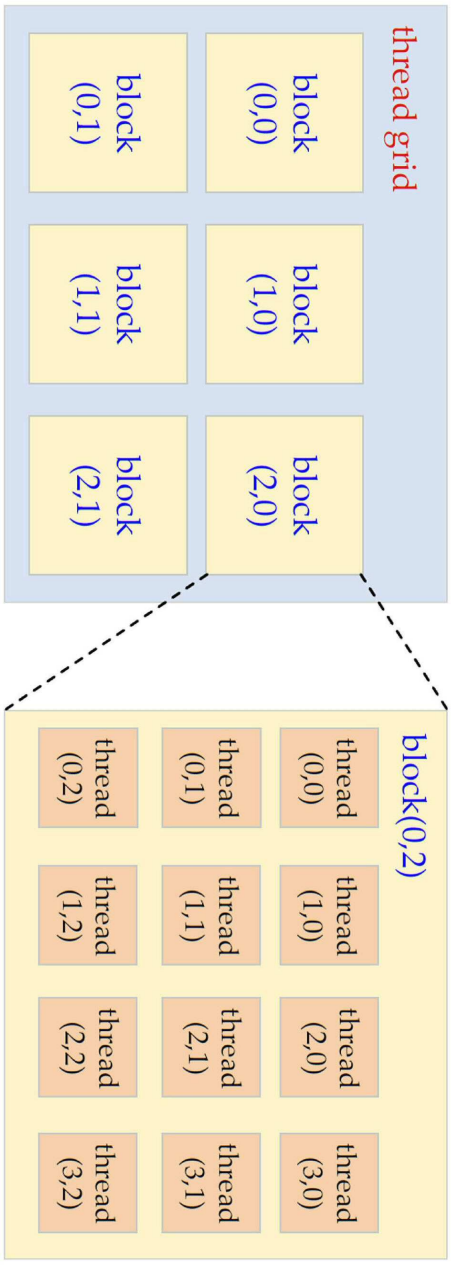

20140401
20140514
20140629
20140715
20140923
Average bitrate
of the five days
7.30424
7.12946
7.19884
7.05144
6.58919
6.42026
6.24072
6.10486
6.16073
5.96426
6.04142
5.83785
bitrate (bpp)
M

(b)


20140401
20140421
20140730
20140824
20140930
Average bitrate
of the five days
7.13847
7.11595
6.96347
6.87259
6.8244
6.74611
6.7575
6.69511
6.10367
6.0454
5.93936
5.87871
bitrate (bpp)
T

(c)

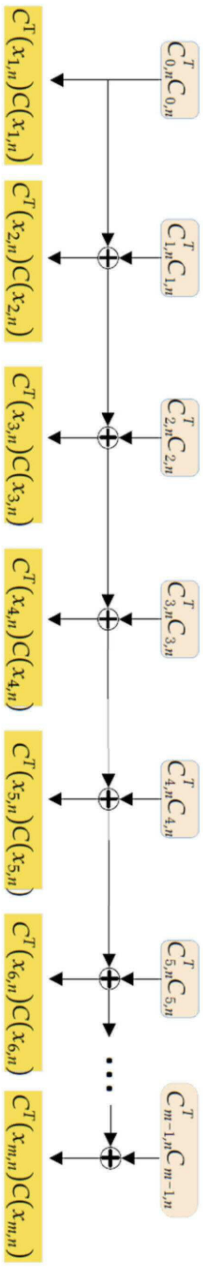
도면9


thread grid
block (0,0)
block (1,0)
block (2,0)
block (0,1)
block (1,1)
block (2,1)
block(0,2)
thread (0,0)
thread (0,1)
thread (0,2)
thread (1,0)
thread (1,1)
thread (1,2)
thread (2,0)
thread (2,1)
thread (2,2)
thread (3,0)
thread (3,1)
thread (3,2)

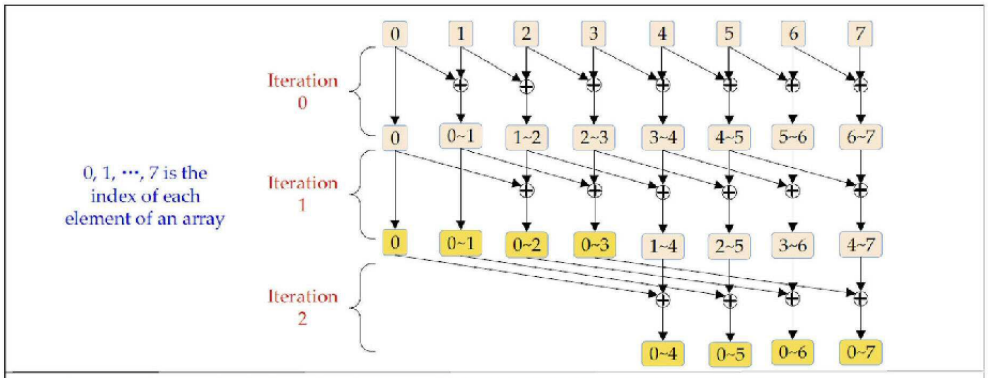
도면10

A small part of an aurora spectral image:
Suppose the prediction order N = 3, and the M = 2.

도면11

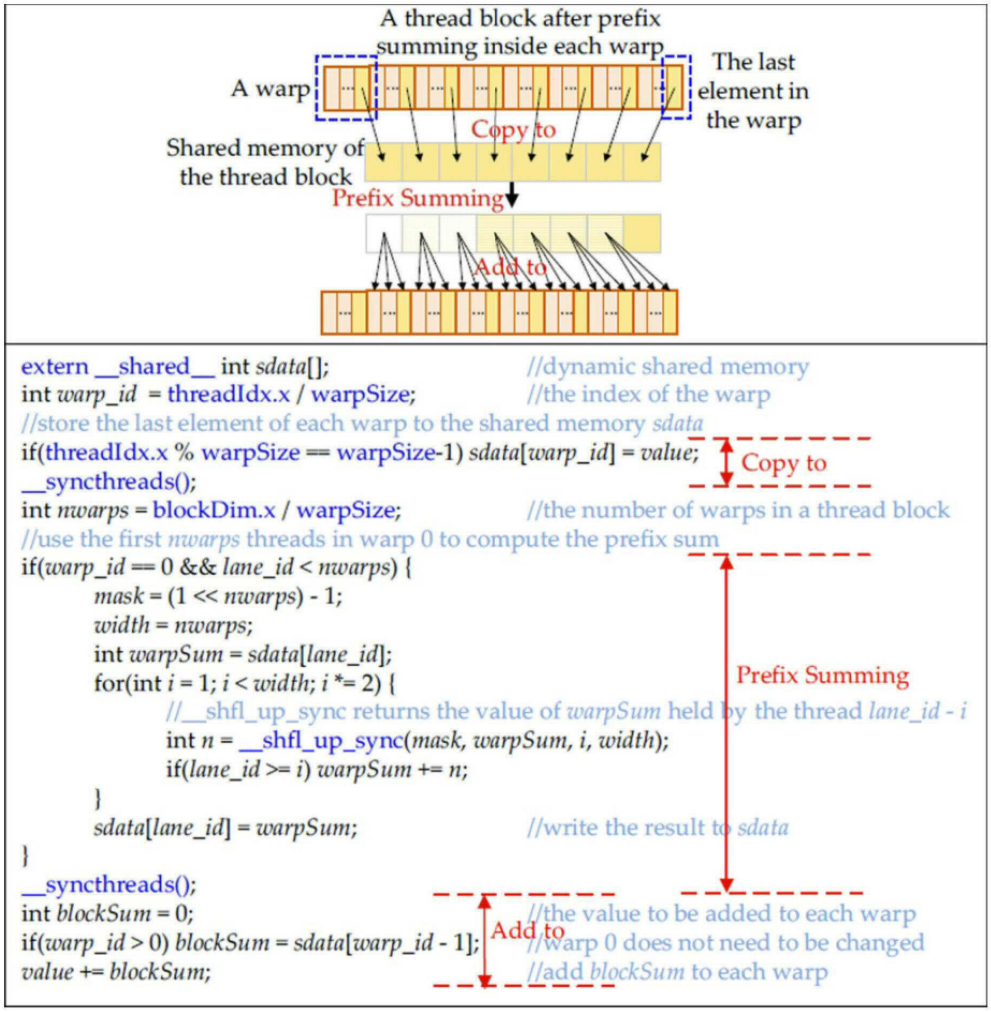

$C^T(x_{1,n})C(x_{1,n})$
$C^T(x_{2,n})C(x_{2,n})$
$C^T(x_{3,n})C(x_{3,n})$
$C^T(x_{4,n})C(x_{4,n})$
$C^T(x_{5,n})C(x_{5,n})$
$C^T(x_{6,n})C(x_{6,n})$
$C^T(x_{m,n})C(x_{m,n})$
$C^T_{0,n}C_{0,n}$
$C^T_{1,n}C_{1,n}$
$C^T_{2,n}C_{2,n}$
$C^T_{3,n}C_{3,n}$
$C^T_{4,n}C_{4,n}$
$C^T_{5,n}C_{5,n}$
$C^T_{m-1,n}C_{m-1,n}$

도면12

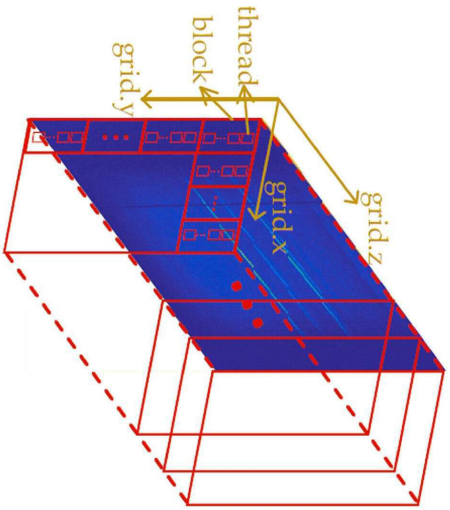



```
int idx = threadIdx.x + blockIdx.x * blockDim.x;  //the index of the thread in the thread grid
if ( idx < N) {                                   //N is the total number of elements
        int value = data[idx];                    //read data from the global memory data[idx]
        //width is the size of the thread group that the shuffle instruction is performed on
        int width = warpSize;                     //warpSize is a built-in variable, which defines the size of a warp
        int lane_id = threadIdx.x % width;        //the index of the thread in the thread group
        unsigned int mask = 0xffffffff;           //indicating the threads participating in the shuffle instruction
        for (int i = 1; i < width; i *= 2){
                //__shfl_up_sync() returns the value of the value held by the thread lane_id - i
                int n = __shfl_up_sync(mask, value, i, width);
                if (lane_id >= i) value += n;
        }
        data[idx] = value;                        //write the result to the global memory data[idx]
}
```

도면13


A thread block after prefix summing inside each warp
A warp
The last element in the warp
Copy to
Shared memory of the thread block
Prefix Summing
Add to
extern __shared__ int sdata[];
//dynamic shared memory
int warp_id = threadIdx.x / warpSize;
//the index of the warp
//store the last element of each warp to the shared memory sdata
if(threadIdx.x % warpSize == warpSize-1) sdata[warp_id] = value;
Copy to
__syncthreads();
int nwarps = blockDim.x / warpSize;
//the number of warps in a thread block
//use the first nwarps threads in warp 0 to compute the prefix sum
if(warp_id == 0 && lane_id < nwarps) {
mask = (1 << nwarps) - 1;
width = nwarps;
int warpSum = sdata[lane_id];
for(int i = 1; i < width; i *= 2) {
//__shfl_up_sync returns the value of warpSum held by the thread lane_id - i
int n = __shfl_up_sync(mask, warpSum, i, width);
if(lane_id >= i) warpSum += n;
}
sdata[lane_id] = warpSum;
//write the result to sdata
Prefix Summing
}
__syncthreads();
int blockSum = 0;
//the value to be added to each warp
if(warp_id > 0) blockSum = sdata[warp_id - 1];
//warp 0 does not need to be changed
value += blockSum;
//add blockSum to each warp
Add to

도면14




```
//each block has nthreads threads
dim3 block(nthreads);
//N is the prediction order
//HEIGHT is the height of an aurora image
//WIDTH is the width of an aurora image
dim3 grid(NxN, (HEIGHT-1+block.x-1)/block.x, WIDTH-2);
```

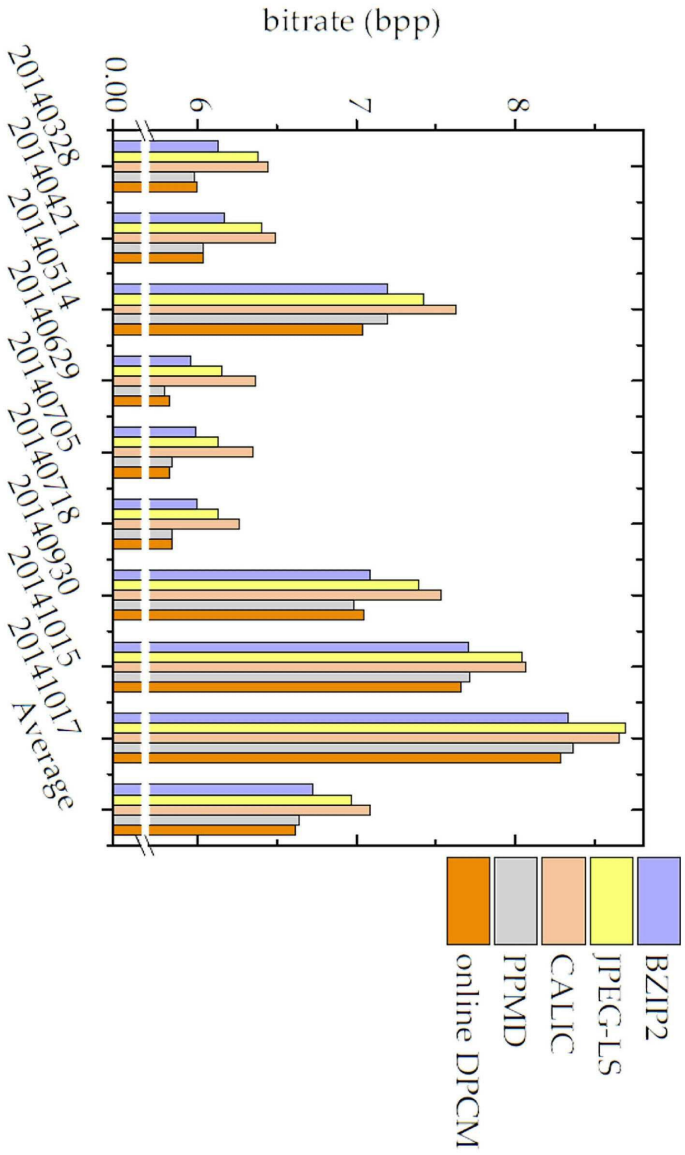
도면15

```
dim3 block(N, N);                    //N is the prediction order, that is, dimension of each matrix C^T_{i,n}C_{i,n}
dim3 grid(WIDTH - 2, HEIGHT - 1);//WIDTH is the width of an aurora image, and HEIGHT is the height
//d_data is the aurora image on the device, d_C^TC is the device memory to store the matrices C^T_{i,n}C_{i,n}
__global__ void matrixMulKernel(unsigned short *d_data, double *d_C^TC) {
        int col = blockIdx.x + 2;        //which column of the aurora image this thread block processes
        int row = blockIdx.y;            //by which row of the aurora image the matrix C_{i,n} is created
        int cols = col;                  //cols is the actual number of columns of C_{i,n}
        /* rows is the actual number of rows of C_{i,n}, that is, how many equations can be built with row row
        of the aurora image */
        int rows = 1;
        if(col > N) {
                cols = N;
                rows = col - N + 1;
                if(rows > M)             //the maximum rows is M
                        rows = M;
        }
        int idx = threadIdx.x + threadIdx.y * blockDim.x;        //the index of this thread in the thread block
        extern __shared__ unsigned short sdata[];
        /* all the elements of C_{i,n} are from a consecutive segment of row i of the aurora image. The
        segment starts from column col-rows+1-cols, ends in column col-1, and the length is cols+rows-1. */
        //copy this segment to the shared memory sdata
        if(idx < cols + rows - 1)    sdata[idx] = d_data[row * WIDTH + col - rows + 1 - cols + idx];
        __syncthreads();
        //the offset of the current C^T_{i,n}C_{i,n} in d_C^TC
        int offset = N * N * (HEIGHT - 1) * blockIdx.x + cols * cols * blockIdx.y;
        if(threadIdx.x < cols && threadIdx.y < cols) {        //the actual size of C^T_{i,n}C_{i,n} is cols*cols
                double sum = 0.0;
                /* each thread is responsible for one element of C^T_{i,n}C_{i,n}, that is, the multiplication of a row
                of C^T_{i,n} and a column of C_{i,n} */
                for(int i = 0; i < rows; i++)    sum += sdata[threadIdx.y + i] * sdata[threadIdx.x + i];
                d_C^TC[offset + threadIdx.y * cols + threadIdx.x] = sum;
        }
}
```

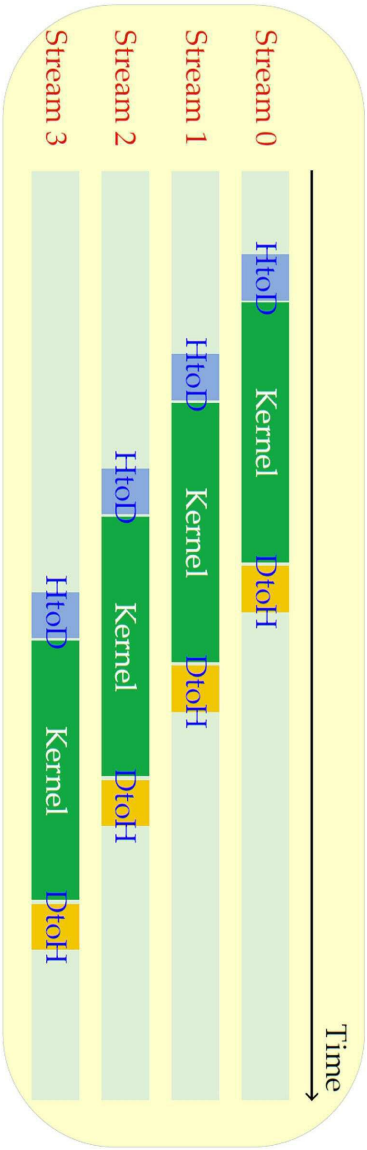
도면16

```
dim3 block(N, N);                    //N is the prediction order, that is, dimension of each matrix C^T_{i,n}C_{i,n}
dim3 grid(WIDTH - 2, HEIGHT - 1);//WIDTH is the width of an aurora image, and HEIGHT is the height
//d_data is the aurora image on the device, d_C^TC is the device memory to store the matrices C^T_{i,n}C_{i,n}
__global__ void matrixMulKernel(unsigned short *d_data, double *d_C^TC) {
        int col = blockIdx.x + 2;        //which column of the aurora image this thread block processes
        int row = blockIdx.y;            //by which row of the aurora image the matrix C_{i,n} is created
        int cols = col;                  //cols is the actual number of columns of C_{i,n}
        /* rows is the actual number of rows of C_{i,n}, that is, how many equations can be built with row row
        of the aurora image */
        int rows = 1;
        if(col > N) {
                cols = N;
                rows = col - N + 1;
                if(rows > M)             //the maximum rows is M
                        rows = M;
        }
        int idx = threadIdx.x + threadIdx.y * blockDim.x;        //the index of this thread in the thread block
        extern __shared__ unsigned short sdata[];
        /* all the elements of C_{i,n} are from a consecutive segment of row i of the aurora image. The
        segment starts from column col-rows+1-cols, ends in column col-1, and the length is cols+rows-1. */
        //copy this segment to the shared memory sdata
        if(idx < cols + rows - 1)    sdata[idx] = d_data[row * WIDTH + col - rows + 1 - cols + idx];
        __syncthreads();
        //the offset of the current C^T_{i,n}C_{i,n} in d_C^TC
        int offset = N * N * (HEIGHT - 1) * blockIdx.x + cols * cols * blockIdx.y;
        if(threadIdx.x < cols && threadIdx.y < cols) {        //the actual size of C^T_{i,n}C_{i,n} is cols*cols
                double sum = 0.0;
                /* each thread is responsible for one element of C^T_{i,n}C_{i,n}, that is, the multiplication of a row
                of C^T_{i,n} and a column of C_{i,n} */
                for(int i = 0; i < rows; i++)    sum += sdata[threadIdx.y + i] * sdata[threadIdx.x + i];
                d_C^TC[offset + threadIdx.y * cols + threadIdx.x] = sum;
        }
}
```

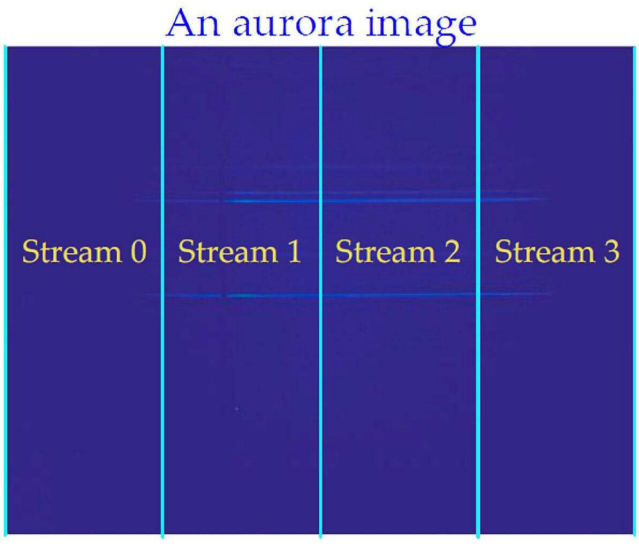
도면17


bitrate (bpp)
0.00
6
7
8
20140328
20140421
20140514
20140629
20140705
20140718
20140930
20141015
20141017
Average
BZIP2
JPEG-LS
CALIC
PPMD
online DPCM

도면18



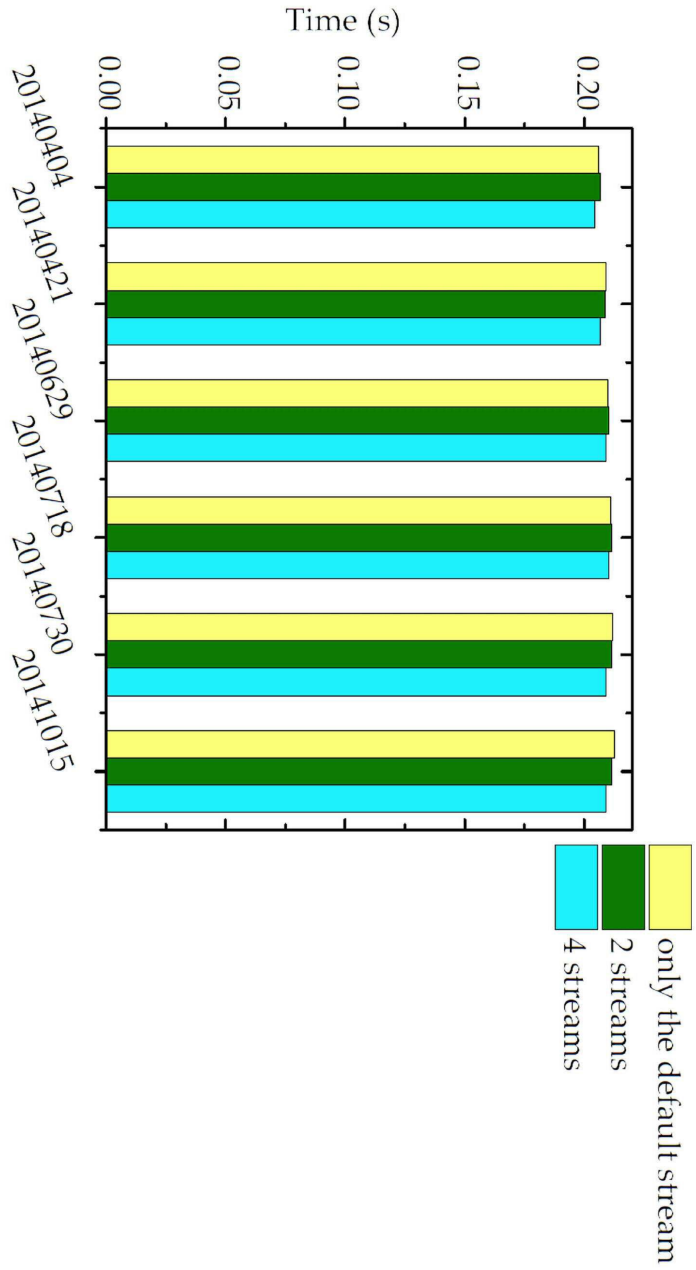
도면19



도면20

```
//NSTREAMS is the number of CUDA streams
cudaStream_t *streams = (cudaStream_t*)malloc(sizeof(cudaStream_t)*NSTREAMS);
//create multiple streams
for (int i = 0; i < NSTREAMS; i++)
        cudaStreamCreate(&streams[i]);
unsigned short *d_auroraData;              //DATASIZE is the number of pixels in an aurora image
cudaMalloc((void**)&d_auroraData, sizeof(unsigned short)*DATASIZE);
short *d_residual;                         //the device memory to store the residuals
cudaMalloc((void**)&d_residual, sizeof(short)*DATASIZE);
//auroraData is the host memory that stores the aurora data
cudaMemcpy(d_auroraData, auroraData, sizeof(unsigned short)*DATASIZE, cudaMemcpyHostToDevice);
for (int i = 0; i < NSTREAMS; i++){
        //the equivalent kernel of all the kernel functions to compute the residuals
        kernel <<<grid, block, size of dynamic shared memory in bytes, streams[i] >>>(d_auroraData, d_residual);
}
//copy the residuals from device to host
cudaMemcpy(residual, d_residual, sizeof(short)*DATASIZE, cudaMemcpyDeviceToHost);
```

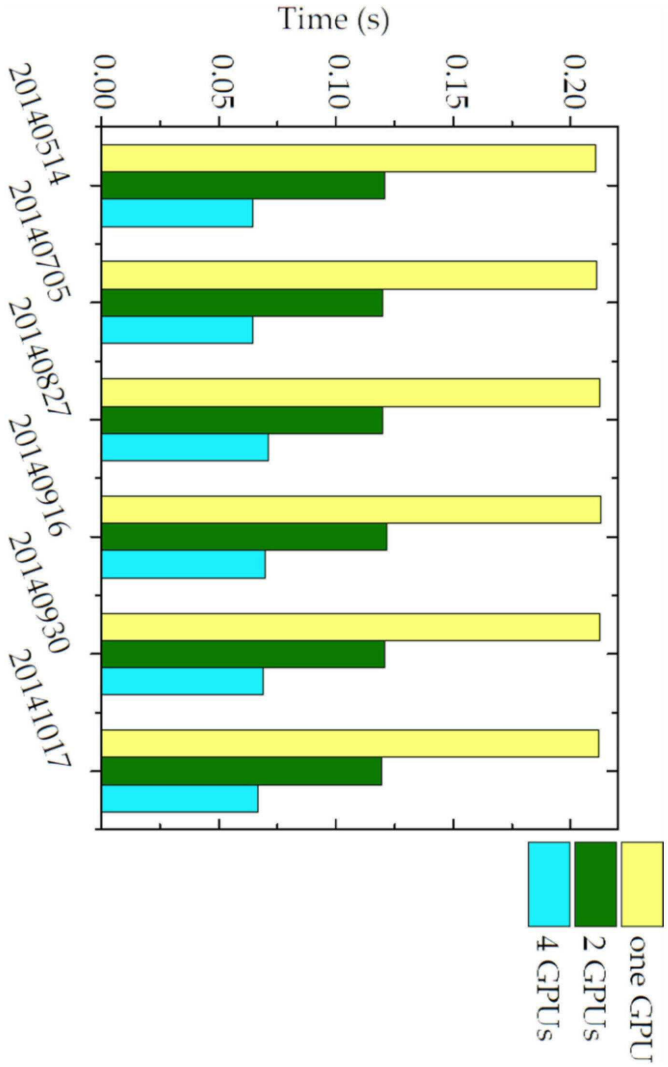
도면21


Time (s)
0.00
0.05
0.10
0.15
0.20
20140404
20140421
20140629
20140718
20140730
20141015
only the default stream
2 streams
4 streams

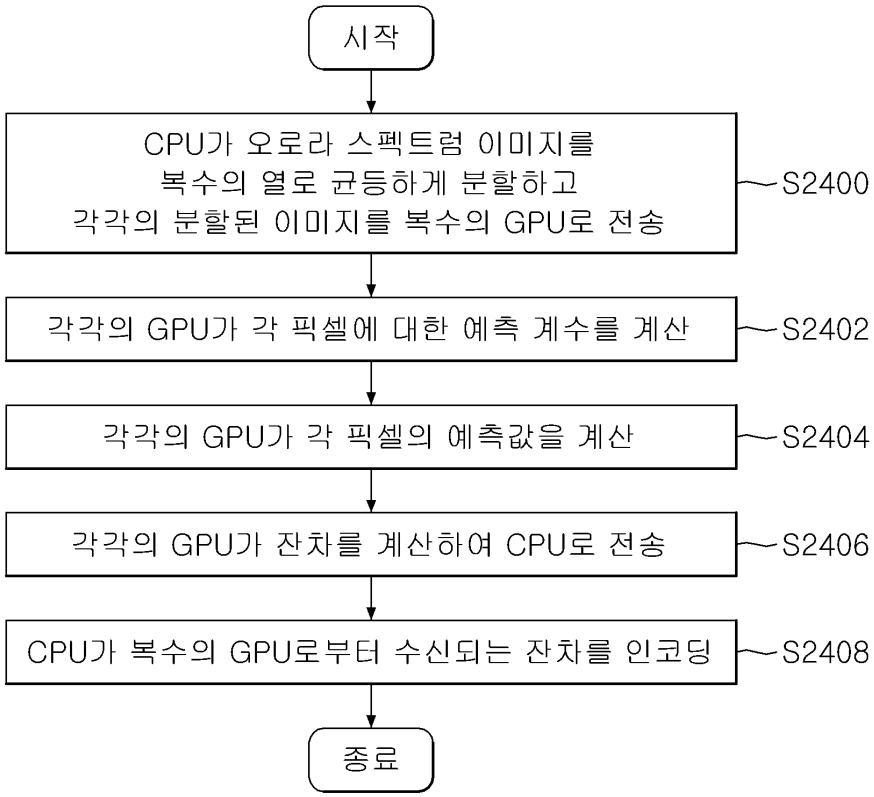
도면22

```
// 1. prepare the aurora data needed by each GPU on the host
//NGPUS is the number of available GPUs
unsigned short**h_auroraData = (unsigned short**)malloc(sizeof(unsigned short*)*NGPUS);
//nColsPerGPU is the number of columns that assigned to each GPU to compute the residual
int nColsPerGPU = WIDTH / NGPUS;      //WIDTH is the number of columns of an aurora image
for (int i = 0; i < NGPUS; i++){
        //all the GPUs except GPU 0 need extra N+M-1 columns of data to process the first column assigned to it
        int dataCols = (i == 0) ? nColsPerGPU : nColsPerGPU + N + M - 1;
        //allocate pinned host memory to store the data for each GPU
        //HEIGHT is the number of rows of an aurora image
        cudaMallocHost((void**)&h_auroraData[i], sizeof(unsigned short)*dataCols*HEIGHT);
        //copy the corresponding aurora data to h_auroraData[i]
        /*.....*/
}
// 2. allocate device memory and create CUDA streams
unsigned short**d_auroraData = (unsigned short**)malloc(sizeof(unsigned short*)*NGPUS);
short**d_residual = (short**)malloc(sizeof(short*)*NGPUS);
cudaStream_t *streams = (cudaStream_t*)malloc(sizeof(cudaStream_t)*NGPUS);
for (int i = 0; i < NGPUS; i++){
        cudaSetDevice(i);                    //set current device
        int dataCols = (i == 0) ? nColsPerGPU : nColsPerGPU + N + M - 1;
        cudaMalloc((void**)&d_auroraData[i], sizeof(unsigned short)*dataCols*HEIGHT);
        cudaMalloc((void**)&d_residual[i], sizeof(short)*nColsPerGPU*HEIGHT);
        cudaStreamCreate(&streams[i]);    //create multiple streams
}
short *h_residual;                           //allocate a pinned memory for asynchronous transfer of residual
cudaMallocHost((void**)&h_residual, sizeof(short)*WIDTH*HEIGHT);
// 3. launch the kernels and asynchronous data transfer
for (int i = 0; i < NGPUS; i++){
        cudaSetDevice(i);
        int dataCols = (i == 0) ? nColsPerGPU : nColsPerGPU + N + M - 1;
        //asynchronous transfer of aurora data from the pinned host memory to each device
        cudaMemcpyAsync(d_auroraData[i], h_auroraData[i], sizeof(unsigned short)*dataCols*HEIGHT,
cudaMemcpyHostToDevice, streams[i]);
        //all the kernel functions are equivalent to this one kernel function
        kernel <<<grid, block, size of dynamic shared memory in bytes, streams[i] >>>(d_auroraData[i], d_residual[i]);
        //asynchronous transfer of residual from each device to the pinned host memory
        cudaMemcpyAsync(h_residual + i*nColsPerGPU*HEIGHT, d_residual[i], sizeof(short)*nColsPerGPU*HEIGHT,
cudaMemcpyDeviceToHost, streams[i]);
}
// 4. synchronize streams
for (int i = 0; i < NGPUS; i++){
        cudaSetDevice(i);
        cudaStreamSynchronize(streams[i]);
}
```

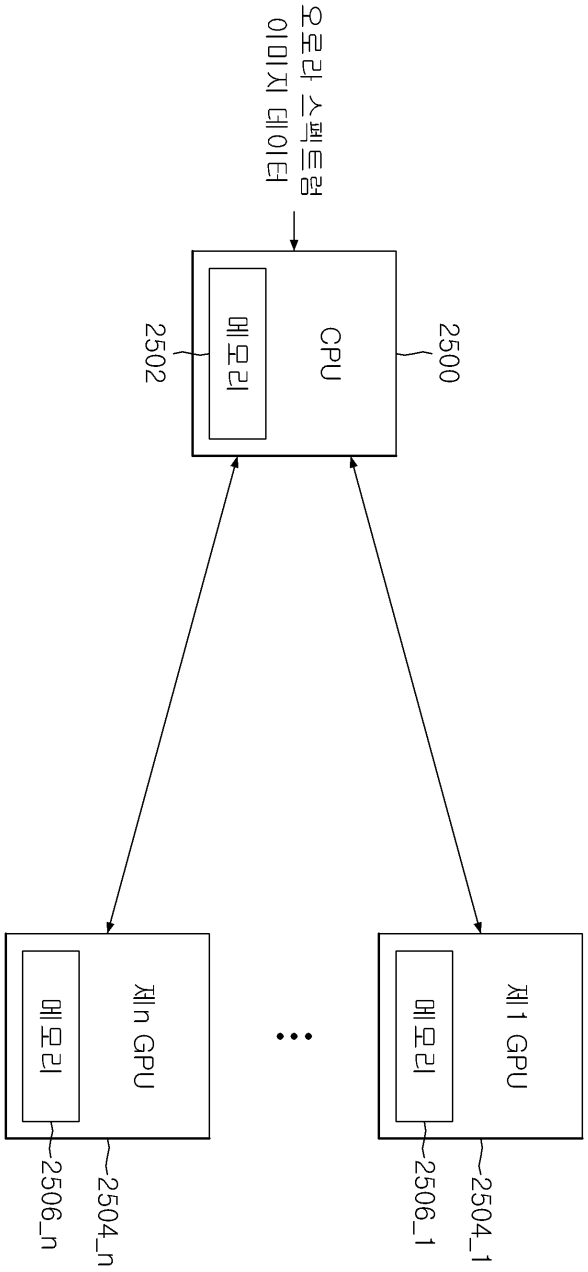
도면23


Time (s)
0.00
0.05
0.10
0.15
0.20
20140514
20140705
20140827
20140916
20140930
20141017
one GPU
2 GPUs
4 GPUs

도면24


시작
CPU가 오로라 스펙트럼 이미지를
복수의 열로 균등하게 분할하고
각각의 분할된 이미지를 복수의 GPU로 전송
S2400
각각의 GPU가 각 픽셀에 대한 예측 계수를 계산
S2402
각각의 GPU가 각 픽셀의 예측값을 계산
S2404
각각의 GPU가 잔차를 계산하여 CPU로 전송
S2406
CPU가 복수의 GPU로부터 수신되는 잔차를 인코딩
S2408
종료

도면25


오로라 스펙트럼
이미지 데이터
CPU
메모리
2500
2502
제1 GPU
메모리
2504_1
2506_1
제n GPU
메모리
2504_n
2506_n