



(19) 대한민국특허청(KR)
(12) 등록특허공보(B1)

(45) 공고일자 2016년08월31일
(11) 등록번호 10-1652264
(24) 등록일자 2016년08월24일

- (51) 국제특허분류(Int. Cl.)
H04N 7/01 (2006.01) H04N 19/112 (2014.01)
H04N 19/16 (2014.01)
- (52) CPC특허분류
H04N 7/01 (2013.01)
H04N 19/112 (2015.01)
- (21) 출원번호 10-2015-0122359
(22) 출원일자 2015년08월31일
심사청구일자 2015년08월31일
- (56) 선행기술조사문헌
Jiaji Wu et al., Parallel Implementation of Edge-Directed Image Interpolation on a Graphics Processing Unit, IEEE, 1052-1056p, 2011.12.07
김민기 외 1인, 새로운 에지 방향 보간법을 이용한 효율적인 디인터레이싱 알고리즘, 방송공학회 논문지, 12권, 제2호, 185-192, 2007.03.
KR1020090090550 A
KR101513395 B1

- (73) 특허권자
인천대학교 산학협력단
인천광역시 연수구 아카데미로 119 (송도동)
- (72) 발명자
전광길
인천광역시 연수구 경원대로119번길 21, 114동 904호 (동춘동, 풍림2차아파트)
- (74) 대리인
특허법인충정

전체 청구항 수 : 총 9 항

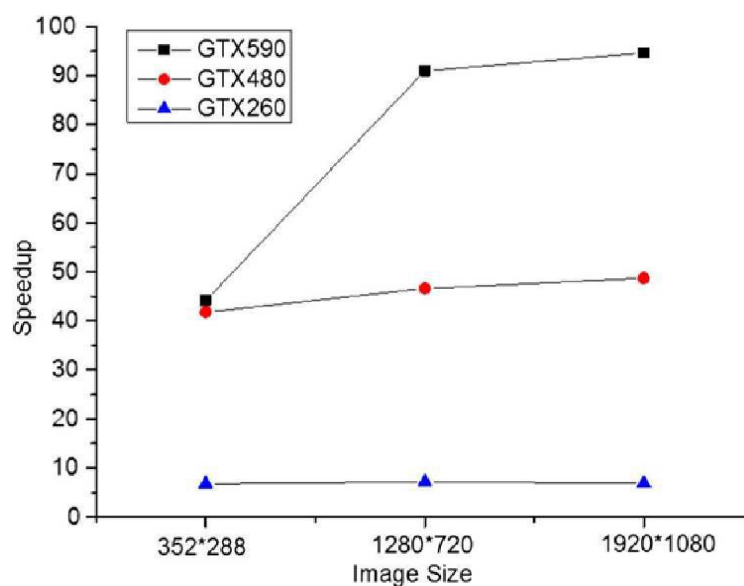
심사관 : 김희주

(54) 발명의 명칭 에지 방향 적응적인 인트라 필드 디인터레이싱 기법을 실행하는 그래픽 프로세싱 유닛의 병렬 구현 방법

(57) 요약

본 발명의 실시예에 따른 에지 방향 적응적인 인트라 필드 디인터레이싱을 실행하는 그래픽 프로세싱 유닛 병렬 구현 방법은, 인터레이스된 이미지로부터 지리적 이중성을 기반으로 디인터레이스 예측 모델을 추정하는 단계 및 상기 디인터레이스 이미지 내의 손실 픽셀들을 보간하는 단계를 포함하고, 상기 디인터레이스 이미지에서 하나의 손실 픽셀을 계산하기 위해 그래픽 프로세싱 유닛이 하나의 쓰레드를 사용한다.

대표도 - 도10



(52) CPC특허분류

H04N 19/16 (2015.01)

H04N 7/012 (2013.01)

공지예외적용 : 있음

명세서

청구범위

청구항 1

에지방향 적응적 인트라 필드 디인터레이싱을 실행하는 그래픽 프로세싱 유닛의 병렬 구현 방법에 있어서,

인터레이스된 이미지로부터 지리적 이중성을 기반으로 디인터레이스 예측 모델을 추정하는 단계; 및

디인터레이스 이미지 내의 손실 픽셀들을 보간하는 단계;를 포함하고,

상기 디인터레이스 이미지에서 하나의 손실 픽셀을 계산하기 위해 그래픽 프로세싱 유닛이 하나의 쓰레드를 사용하여,

상기 디인터레이스 예측 모델을 생성하기 위해, 보간될 픽셀 주위의 6개의 이웃 픽셀들을 사용하고, 상기 인터레이스 이미지 X의 크기가 H x W이고, 디인터레이스 이미지 Y의 크기가 2H x W일 때, 상기 디인터레이스 이미지에 있는 픽셀 $y_{2i+1, j}$ 는 상기 인터레이스 이미지로부터의 픽셀 $x_{i, j}$ 과 같고, 상기 지리적 이중성의 관계에 의해 상기 보간된 픽셀은

$$y_{2i+1, j} = \sum_{m=0}^1 \sum_{n=0}^2 \alpha_{3m+n}^h x_{2(i+m), j-1+n}$$

로 추정되며, 여기서 $y_{2i+1, j}$ 는 디인터레이스 이미지에서 보간

될 픽셀이고, $x_{2(i+m), j-1+n}$ ($m=0, 1; n=0, 1, 2$)은 인터레이스된 알려진 픽셀들이며, α_{3m+n}^h ($m=0, 1; n=0, 1, 2$)은 가중치이고, 상기 가중치 값은 픽셀 $y_{2i+1, j}$ 에 대응하는 계수 $x_{2(i+m), j-1+n}$ ($m=0, 1; n=0, 1, 2$)의 효과를 나타내는 것을 특징으로 하는, 에지 방향 적응적인 인트라 필드 디인터레이싱 기법의 그래픽 프로세싱 유닛 병렬 구현 방법.

청구항 2

제1항에 있어서,

초기에 상기 인터레이스 이미지와 중간 변수들을 상기 그래픽 프로세싱 유닛의 공유 메모리에 저장하고, 각각의 상기 쓰레드는 상기 공유 메모리로부터 데이터를 가져오며, 상기 공유 메모리는 다수 개의 뱅크들로 나누어지고, 상기 쓰레드들이 다른 뱅크들로부터 동시에 데이터를 접근할 수 있게 하기 위해, 하나의 블록 내에 있는 모든 상기 쓰레드들에 대해 동기화를 실행하는 단계를 더 포함하는 것을 특징으로 하는, 에지 방향 적응적인 인트라 필드 디인터레이싱 기법의 그래픽 프로세싱 유닛 병렬 구현 방법.

청구항 3

제1항에 있어서,

상기 디인터레이스 예측 모델을 추정하는 단계 및 상기 디인터레이스 이미지 내의 손실 픽셀들을 보간하는 단계에서 자동으로 생성된 일부 변수들을 상기 그래픽 프로세싱 유닛의 레지스터 메모리에 저장하고, 상기 레지스터 메모리는 특정 쓰레드들에 속하는 것을 특징으로 하는, 에지 방향 적응적인 인트라 필드 디인터레이싱 기법의 그래픽 프로세싱 유닛 병렬 구현 방법.

청구항 4

제1항에 있어서,

상기 그래픽 프로세싱 유닛 장치와 호스트 장치 간의 데이터 전송에 있어서, 비동기 데이터 전송(ADT)을 채택하고, 하나의 스트림이 연산을 위해 커널을 실행할 수 있도록 허용하는 동안, 두 번째 스트림이 호스트 장치와 그래픽 프로세싱 유닛 장치사이의 데이터 전송을 실행하도록 허용하는 것을 특징으로 하는, 에지 방향 적응적인 인트라 필드 디인터레이싱 기법의 그래픽 프로세싱 유닛 병렬 구현 방법.

청구항 5

제1항에 있어서,

상기 디인터레이싱을 위해 다중 그래픽 프로세싱 유닛을 이용하는 것을 특징으로 하는, 에지 방향 적응적인 인트라 필드 디인터레이싱 기법의 그래픽 프로세싱 유닛 병렬 구현 방법.

청구항 6

제1항에 있어서, 상기 지리적 이중성은,

다른 해상도에서 같은 방향에 따른 픽셀들의 쌍을 짓는 것으로서, 상기 지리적 이중성을 이용하여 에지 방향 정보를 감지하는 것을 특징으로 하는, 에지 방향 적응적인 인트라 필드 디인터레이싱 기법의 그래픽 프로세싱 유닛 병렬 구현 방법.

청구항 7

삭제

청구항 8

제1항에 있어서,

상기 지리적 이중성의 사용을 통해 인터레이스 가중치 $\overrightarrow{\alpha^l} = [\alpha_0^l, \alpha_1^l, \alpha_2^l, \alpha_3^l, \alpha_4^l, \alpha_5^l]^T$ 를 풀어서 디인터레이스 가중치 $\overrightarrow{\alpha^h} = [\alpha_0^h, \alpha_1^h, \alpha_2^h, \alpha_3^h, \alpha_4^h, \alpha_5^h]^T$ 를 해결하는 것을 특징으로 하는, 에지 방향 적응적인 인트라 필드 디인터레이싱 기법의 그래픽 프로세싱 유닛 병렬 구현 방법.

청구항 9

제8항에 있어서,

상기 인터레이스 가중치 $\overrightarrow{\alpha^l} = [\alpha_0^l, \alpha_1^l, \alpha_2^l, \alpha_3^l, \alpha_4^l, \alpha_5^l]^T$ 를 풀기 위해 상기 인터레이스 이미지 내의 로컬 윈도우 내에서 인터레이스 이미지 내의 로컬 윈도우 내에서 M ($M \geq 6$) 예측 모델을 생성하고, 상기 인자 $\overrightarrow{\alpha^l}$ 는
$$\overrightarrow{\alpha^l} = \underset{\overrightarrow{\alpha^l}}{\operatorname{argmin}} \sum_{i=1}^n \|x_i - \sum_{t=1}^6 x_{i-t} \alpha_t\|$$
 의 선형 사각형 추정을 통해 획득되며, 여기서 x_i 는 상기 인터레이스 이미지 안의 픽셀이고, x_{i-t} 는 상기 픽셀 x_i 의 6개의 연결된 이웃 픽셀들인 것을 특징으로 하는, 에지 방향 적응적인 인트라 필드 디인터레이싱 기법의 그래픽 프로세싱 유닛 병렬 구현 방법.

청구항 10

제9항에 있어서,

상기 인자 $\overrightarrow{\alpha^l}$ 는
$$\overrightarrow{\alpha^l} = \underset{\overrightarrow{\alpha^l}}{\operatorname{argmin}} \|\overrightarrow{x} - C\overrightarrow{\alpha}\|,$$
 의 행렬 형태로 쓰여지고, 여기서 $\overrightarrow{x} = [x_1, x_2, \dots, x_n]^T$ 는 가중치 $\overrightarrow{\alpha}$ 를 정교하게 하는데 사용되는 $M \times (M+1)$ 인터레이스 이미지 픽셀을 포함하는 벡터이며, $C = [C_0, C_1, \dots, C_n]^T$ 은 k 번째 열 벡터 $C_k = [C_{0k}, C_{1k}, C_{2k}, C_{3k}, C_{4k}, C_{5k}]$ 가 x_k 주위의 가장 가까운 6개의 이웃들을 포함하는 $M \times (M+1) \times 6$ 데이터 행렬이고, 알파들은 상기 인자에 관한 수학적식을 최소화해서 $\overrightarrow{\alpha^l} = (C^T C)^{-1} C^T \overrightarrow{x}$ 로 얻고, 상기 지리적 이중성에 따라, $\overrightarrow{\alpha^h} \cong \overrightarrow{\alpha^l}$ 방정식을 얻는 것을 특징으로 하

는, 예지 방향 적응적인 인트라 필드 디인터레이싱 기법의 그래픽 프로세싱 유닛 병렬 구현 방법.

발명의 설명

기술 분야

- [0001] 본 발명은 그래픽 프로세싱 유닛의 병렬 구현 방법에 관한 것으로, 특히 인터레이스 공분산과 디인터레이스 공분산 사이에서의 지리적 이원성에 따른, 인터레이스 공분산으로부터 추정된 디인터레이스 공분산에 근거한 손실 픽셀을 보간하는 예지 방향 적응적인 인트라 필드 디인터레이싱 기법에 있어서, 효과적인 그래픽 프로세싱 유닛 기반의 견고한 병렬 구현 방법에 관한 것이다.

배경 기술

- [0002] 대부분의 방송 TV 시스템에서, NTSC, PAL, SECAM과 같이 비디오 전송 대역폭을 반으로 줄일 수 있는 인터레이스 스캔 포맷들은 실제 응용 사례에서 여전히 널리 사용되고 있다. 그러나 인터레이스 스캔 포맷들은, 내측의 깜빡임, 얼룩, 필드 에일리어싱(field aliasing)과 같은 약간의 인공 결점들이 발생할 수 있으며, 이들은 인터레이스 이미지의 가시 품질을 심하게 떨어트린다. 특히, DTV 수신기의 개발에 따라, 디인터레이싱은 DTV에서 수용된 다양한 표준 포맷들을 지원하는 필수적인 요소가 되었고, 그것의 성능은 수신기로부터 나타나는 최종 이미지 품질에서의 주요 요소가 되었다.
- [0003] 디인터레이싱은 두 종류를 포함하는데, 그것은 인트라 필드(intra-field) 부분 도메인 기법들과, 인터 필드(inter-field)의 일시적인 도메인 기법들이다. 비록 일시적인 도메인 기법들이, 부분 도메인 기법들보다 더 나은 성능을 보여주어도, 그것들의 구현에는 많은 양의 연산과 메모리를 요구한다. 반대로, 보간의 분야에 대한 정보만 사용하는 인트라 필드 공간 방식들은 실제 응용에서 널리 쓰여진다.
- [0004] 다양한 인트라 필드 디인터레이싱 방식들은, 연산 복잡도와 재생성 품질 사이에서 좋은 균형을 유지하기 위해, 개발되어 왔다. 예를 들면, 라인-평균(line-averaging: LA) 알고리즘은, 위아래 픽셀들의 평균값을 사용하여 픽셀을 보간한다.
- [0005] 비록 선형 인트라 필드 디인터레이싱 알고리즘들이 낮은 연산 복잡도를 가지지만 그들은 디인터레이스 이미지에서 수직 방향의 선명도가 감소되는 흐릿한 윤곽선의 경향을 가진다. 이 문제를 해결하기 위해, 많은 우수한 인트라 필드 디인터레이싱 방식들이 섬세함을 보존하고 윤곽선의 선명도를 유지하기 위해 제안되어 왔다. 이들 중, 가장 유망한 것은, Lee에 의해 제안된, 공분산 기반 적응적 디인터레이싱(covariance-based adaptive deinterlacing) 방식이며, 그는 디인터레이싱 분야에 Li의 근본적인 아이디어를 직접 적용했다.

발명의 내용

해결하려는 과제

- [0006] 본 발명의 목적은 디인터레이스 이미지에서 각각의 손실 픽셀을 추정하기 위해 단일 쓰레드를 사용하고, 그래픽 프로세싱 유닛에 존재하는 수백만의 쓰레드를 이용하기 위해 메모리 대역폭을 개선하는 최적화 기법들을 제안하는, 예지 방향 적응적인 인트라 필드 디인터레이싱 기법을 실행하는 그래픽 프로세싱 유닛의 병렬 구현 방법을 제공하는데 있다.

과제의 해결 수단

- [0007] 상기한 기술적 과제를 달성하기 위한 본 발명의 실시예에 따른 예지 방향 적응적인 인트라 필드 디인터레이싱 기법을 실행하는 그래픽 프로세싱 유닛의 병렬 구현 방법은, 인터레이스된 이미지로부터 지리적 이중성을 기반으로 디인터레이스 예측 모델을 추정하는 단계 및 상기 디인터레이스 이미지 내의 손실 픽셀들을 보간하는 단계를 포함하고, 상기 디인터레이스 이미지에서 하나의 손실 픽셀을 계산하기 위해 그래픽 프로세싱 유닛이 하나의 쓰레드를 사용한다.

발명의 효과

- [0008] 본 발명의 실시예에 따른 예지 방향 적응적인 인트라 필드 디인터레이싱 기법을 실행하는 그래픽 프로세싱 유닛의 병렬 구현 방법에 의하면, 레지스터와 공유 메모리 및 L1 캐시를 이용함으로써 신속한 메모리 접근을 구현하고, 비동기 데이터 전송을 실행함으로써 호스트 장치와 GPU 장치 간의 전송시간을 줄일 수 있다.

도면의 간단한 설명

- [0009] 도 1은 본 발명의 지리적 이중성의 관계를 도시한다.
- 도 2는 본 발명의 GPU 디바이스 구조를 도시한다.
- 도 3은 본 발명의 L1 캐쉬와 분배된 메모리의 메모리 분배도를 도시한다.
- 도 4는 본 발명의 데이터 전송의 비동기 아웃라인을 도시한다.
- 도 5는 본 발명의 멀티 GPU 실행을 도시한다.
- 도 6은 속도 비교를 나타내는 그래프를 도시한다.
- 도 7(a)~9(a)는 원본 이미지를 도시한다.
- 도 7(b)~9(b)는 인터레이스 이미지를 얻기 위한 샘플 이미지를 도시한다.
- 도 7(c)~9(c)는 GPU에 의해 디인터레이스된 이미지를 도시한다.
- 도 7(d)~9(d)는 CPU의 직렬 코드로 디인터레이스된 이미지들을 도시한다.
- 도 10은 본 발명의 각각 다른 그래픽 프로세싱 유닛에 대해, 다른 비디오 이미지들에 대한 다른 성능을 도시하는 그래프이다.

발명을 실시하기 위한 구체적인 내용

- [0010] 본 발명을 충분히 이해하기 위해서 본 발명의 바람직한 실시예를 첨부 도면을 참조하여 설명한다. 본 발명의 실시예는 여러 가지 형태로 변형될 수 있으며, 본 발명의 범위가 아래에서 상세히 설명하는 실시예로 한정되는 것으로 해석되어서는 안 된다. 본 실시예는 당업계에서 평균적인 지식을 가진 자에게 본 발명을 보다 완전하게 설명하기 위하여 제공 되는 것이다. 따라서 도면에서의 요소의 형상 등은 보다 명확한 설명을 강조하기 위해서 과장되어 표현될 수 있다. 각 도면에서 동일한 부재는 동일한 참조부호로 도시한 경우가 있음을 유의하여야 한다. 또한, 본 발명의 요지를 불필요하게 흐릴 수 있다고 판단되는 공지 기능 및 구성에 대한 상세한 기술은 생략된다.
- [0011] 본 발명의 예지 방향 적응적 인트라 필드 디인터레이싱 방식은 반복적이지 않은 오리엔테이션-적응적(orientation-adaptive) 방식이며, 이는 기존의 인트라 디인터레이싱(intra-deinterlacing) 방식들 보다 더 나은 주관적인 품질을 가지는 디인터레이스 이미지를 달성하기 위한 것이다. 이를 위해 예지 방향 추정 대신, 지리적 이원성을 사용하는 인터레이스 공분산으로부터 디인터레이스 공분산의 추정을 통해 손실 픽셀을 보간한다. 따라서 각각의 예지 방향은 그 자신의 보간 계수를 가지며, 디인터레이스 이미지의 손실 픽셀들은 다른 계수들을 사용하여 추정된다.
- [0012] 이하 본 발명의 상세한 설명에 앞서, 발명의 배경에 대해 기술하기로 한다.
- [0013] 비록 우수한 가시 품질을 얻을지도 모르는 몇몇 인트라 필드 디인터레이싱 기법들이 최근에 제안되었지만, 많은 수의 유사한 기법들이 위에 언급한 기법에 기반한 개선으로부터 나온다. 따라서, 본 발명의 예지 방향 적응적 인트라 필드 디인터레이싱 기법은 대표적이고 결정적인 인트라 필드 디인터레이싱 알고리즘이다. 하지만 공분산 기반 적응적 디인터레이싱 기법은 각각의 보간된 픽셀에 대해, 행렬 곱셈, 행렬 전환, 역행렬 등과 같은 많은 행렬 연산을 수반한다. 시간 중요도가 높은 시나리오에서 디인터레이싱 기법의 실행을 지연시키는 수백만 픽셀을 가지는 큰 이미지에서 특히, 행렬 연산이 비용이 크다는 것은 의심할 여지가 없다. 그것의 인상적인 성능에도 불구하고, 높은 연산 복잡도의 제한 때문에 공학 분야에서의 많은 응용 사례를 찾기는 힘들다.
- [0014] CPU에서의 연산 영역에서, 하나의 반복문에서 손실 픽셀이 보간된다는 것을 볼 수 있다. 하지만 이러한 반복문은 그래픽 프로세싱 유닛에서의 수많은 쓰레드들을 사용하는 연산으로 교체될 수 있다. 왜냐하면 각각의 반복문은 독립적으로 실행이 가능하기 때문이다. 달리 말하면, 하나의 쓰레드는 본 발명의 병렬 디인터레이싱 기법에서 하나의 손실 픽셀을 계산하는 것이다. 그래픽 프로세싱 유닛 아키텍처에서의 경이적인 개발로 공분산 기반 적응적 디인터레이싱 기법의 연산 복잡도는 그래픽 프로세싱 유닛의 막대한 연산 수용력의 이점으로 인해 줄어들 수 있다. 게다가, 그래픽 프로세싱 유닛 기반 병렬 인트라 필드 디인터레이싱 기법은 약간의 수정만으로 다른 유사한 디인터레이싱 기법들에 적용될 수 있다.

- [0015] 최근에 그래픽 프로세싱 유닛은 수천 개의 코어를 가지는 다중 쓰레드, 높은 연산 가능성, 그리고 매우 높은 메모리 대역폭으로 발달되었다. 그들은 그래픽 렌더링 연산의 속도를 높이는 것뿐만 아니라, 일반적인 목적의 과학적이고 공학적인 응용들을 가속하는데도 사용된다.
- [0016] 그래픽 프로세싱 유닛의 연산 수용력이 증가함에 따라, 더 많은 과학자들과 공학자들이 비용 효율적이고 높은 성능의 연산 플랫폼으로 그것들을 이용한다. 그래픽 프로세싱 유닛은 일반적인 목적의 슈퍼컴퓨팅, 높은 병렬성, 높은 메모리 대역폭, 저비용, 작은 크기와 같은 많은 중요한 특징들을 가지며, 그러한 특징들은 그래픽 프로세싱 유닛 기반 데스크탑 컴퓨터가, CPU들로 만들어진 병렬 시스템과 비교하여 더 매력적인 후보가 되게 한다.
- [0017] 2006년 11월에 GeForce 8800 GTX는 NVIDIA의 CUDA로 만들어진 최초의 그래픽 프로세싱 유닛이다. CUDA는 새로운 병렬 프로그래밍 모델과 명령어 집합 아키텍처를 가지는 일반적인 목적의 병렬 컴퓨팅 아키텍처이다. CUDA는 그래픽 프로세싱 유닛이 복잡한 연산 문제의 해결 및 그래픽 프로세싱 유닛 프로그래밍을 단순화 시켜준다. CUDA 어플리케이션은 CPU에서 동작하는 직렬 프로그램과 그래픽 프로세싱 유닛에서 동작하는 병렬 프로그램의 두 부분으로 구성된다. 병렬부는 커널(kernel)이라 불리며, 이는 동시 실행을 위해, 그래픽 프로세싱 유닛 코어들에서 수백만의 쓰레드를 생성할 수 있다.
- [0018] 그래픽 프로세싱 유닛은 다양한 영역에서 수치적인 연산 문제를 다루는데서 매우 성공적으로 적용되어 왔다. Kowarzyk은 디인터레이싱 기법의 병렬 실행을 보여주었다. 그것은 PBDI 디인터레이싱 알고리즘의 GP그래픽 프로세싱 유닛 기반 소프트웨어 실행이다. 저자는 패턴 기반 방향성 인트라 필드 디인터레이싱 알고리즘을 그래픽 프로세싱 유닛에 성공적으로 적용시켰으며, 그것은 그래픽 프로세싱 유닛 플랫폼에서의 인트라 필드 디인터레이싱(inter-field deinterlacing)의 실행을 강조한다.
- [0019] 그들의 연구에서, 그래픽 프로세싱 유닛 실행은 초당 237개의 고선명도 부분을 디인터레이스 할 수 있었으며, 이에 비해, CPU 실행은 초당 3.47개의 고선명도 부분을 디인터레이스 했다. 저자는 인텔 Xeon E5430에 비하여, NVIDIA Tesla C1060 그래픽 프로세싱 유닛에서 68-fold 속도 향상을 달성했다.
- [0020] 그래픽 프로세싱 유닛 하드웨어와 CUDA 프로그래밍에서의 최근의 진보의 주류에 염감을 얻어, 본 발명은 에지 방향 적응적 인트라 필드 디인터레이싱 방법의 견고한 병렬 실행이 NVIDIA GeForce GTX 590 그래픽 프로세싱 유닛에서 동작하도록 개발했다. 비록 각 픽셀의 주변 맥락이 주위 픽셀들의 그것과 겹쳐지지만, 에지 방향 적응적 인트라 필드 디인터레이싱 방법에서 디인터레이스 이미지들에서의 손실 픽셀들은 독립적으로 추정된다. 따라서, 이 인트라 디인터레이싱 알고리즘은 그래픽 프로세싱 유닛에서 실행될 수 있다.
- [0021] 먼저, 요약적으로 본 발명에서는 에지 방향 적응적 인트라 필드 디인터레이싱 기법을 실행하는 그래픽 프로세싱 유닛의 병렬 구현 방법이 제안되었다. 제안된 방법의 기본 아이디어는 디인터레이스 이미지에서 각각의 손실 픽셀을 추정하기 위해 단일 쓰레드를 사용하는 것이다. 그래픽 프로세싱 유닛에 존재하는 수백만의 쓰레드를 이용하기 위해, 본 발명은 메모리 대역폭을 개선하는 최적화 기법들에 집중하며, 따라서, 더 많은 구현 세부내용이 언급된다. 결과는 에지 방향 적응적 인트라 필드 디인터레이싱 기법의 버전에서, 기존 CPU 구현과 비교하여 디인터레이싱을 실행하는데 94.6배의 속도 향상을 보여주며, NVIDIA GeForce GTX 590 그래픽 프로세싱 유닛에서 동작하도록 배치된다.
- [0022] 이하, 첨부된 도면을 참조하여 본 발명의 바람직한 실시예를 설명함으로써, 본 발명의 에지 방향 적응적 인트라 필드 디인터레이싱 기법을 상세히 설명한다.
- [0023] 에지는 2D 이미지를 처리할 때 중요한 요소이다. 따라서, 좋은 가시 품질을 얻기 위해, 에지의 방향을 정확히 추정하는 것이 중요하다. 에지 방향 적응적 인트라 필드 디인터레이싱 기법에서, 디인터레이스 예측 모델은 지리적 이중성을 기반으로 인터레이스된 이미지로부터 추정된다.
- [0024] 에지 방향을 따르는 매끈함과 에지 방향을 가로지르는 날카로움과 관련된 지리적 균형은 에지의 기본적인 특징 중의 하나이다. 이미지의 에지 특징들은 그것의 공분산을 통해 반영될 수 있다. 지역적 공분산 구조가 에지의 충분한 방향성 정보를 제약한다는 것은 의심의 여지가 없다. 따라서 본 발명은 공분산을 통해 지역적 균형을 묘사하는데 사용되는 상관 계수를 조정할 수 있다.
- [0025] 핵심은 알려진 인터레이스 공분산으로부터 디인터레이스 공분산을 얻는 것이다. 디인터레이스 이미지는 인터레이스 이미지와 유사한 에지 특징들을 가진다. 이는 디인터레이스 공분산과 인터레이스 공분산 사이의 관계를 묘사하는 지리적 이중성으로 언급된다. 단순 용어로, 지리적 이중성은 다른 해상도에서 같은 방향에 따른 픽셀들

의 쌍을 짝짓는다. 디인터레이스 공분산은 지리적 이중성에 의해 인터레이스 공분산으로부터 얻을 수 있다. 이미지의 지리적 이중성은 수많은 이미지 처리 작업들에 널리 이용되어 왔다. 에지 방향 적응적 인트라 필드 디인터레이싱 기법은 이 지리적 이중성을 이용하며, 이것이 기존 인트라 필드 디인터레이싱과 비교하여, 에지 방향 정보를 감지하기 위한 더욱 효율적인 방법인 이유이다.

[0026] 문제를 단순하게 묘사하기 위해, 본 발명은 인터레이스 이미지 X의 크기가 $H \times W$ 이고 디인터레이스 이미지 Y의 크기는 $2H \times W$ 라고 가정한다. 디인터레이스 이미지에 있는 픽셀 $y_{2i+1, j}$ 는 인터레이스 이미지로부터의 픽셀 $x_{i, j}$ 과 같다. 그러면 Y로부터의 인터레이스 라인들 $y_{2i+1, j}$ 는 보간될 것이다.

[0027] 이 발명에서 예측 모델을 생성하기 위해, 보간될 픽셀 주위의 6개의 이웃 픽셀들을 사용한다. 지리적 이중성의 관계는 도 1에 도시된다. 검은 점은 알려진 인터레이스 픽셀이고, 하얀 점은 손실된 픽셀을 나타낸다. 보간된 픽셀은 수학적(1)에서 보여주듯이, sixth-order 가중치 있는 선형 보간에 의해 추정된다.

$$y_{2i+1, j} = \sum_{m=0}^1 \sum_{n=0}^2 \alpha_{3m+n}^h x_{2(i+m), j-1+n} \quad (1)$$

[0029] 여기서 $y_{2i+1, j}$ 는 디인터레이스 이미지에서 보간될 픽셀이고, $x_{2(i+m), j-1+n}$ ($m=0,1; n=0,1,2$)은 인터레이스된 알려진 픽셀들이다. α_{3m+n}^h ($m=0,1; n=0,1,2$)은 이른바 가중치이며, 그것의 값은 픽셀 $y_{2i+1, j}$ 에 대응하는 계수 $x_{2(i+m), j-1+n}$ ($m=0,1; n=0,1,2$)의 효과를 나타낸다. 수학적 (1)을 해결하는 핵심은 계수 벡터 $\overline{\alpha^h} = [\alpha_0^h, \alpha_1^h, \alpha_2^h, \alpha_3^h, \alpha_4^h, \alpha_5^h]^T$ 의 해를 찾는 것이다.

[0030] 디인터레이스 가중치 $\overline{\alpha^h}$ 를 지리적 이중성의 사용을 통해 인터레이스 가중치 $\overline{\alpha^l} = [\alpha_0^l, \alpha_1^l, \alpha_2^l, \alpha_3^l, \alpha_4^l, \alpha_5^l]^T$ 를 풀어서 해결할 수 있다. 본 발명에서 제안된 디인터레이스 방식은, 이미지가 구분적으로 고정된다는 추정에 근거하며, 따라서 디인터레이스 공분산과 인터레이스 공분산 사이에는 관련성이 있다.

[0031] 모두가 알듯이, 자연 이미지는 지역으로 고정된 가우시안(Gaussian) 프로세스들로 보일 수 있다. 따라서, 구분적인 고정은 예측의 손실없는 이미지 압축에서 성공적으로 입증되어온 적절한 추정이다. 따라서 본 발명은 인터레이스 가중치 $\overline{\alpha^l} = [\alpha_0^l, \alpha_1^l, \alpha_2^l, \alpha_3^l, \alpha_4^l, \alpha_5^l]^T$ 를 풀기 위해, 인터레이스 이미지 내의 로컬 윈도우 내에서 M ($M \geq 6$) 예측 모델을 생성할 수 있다.

[0032] 윈도우의 크기 선택은 일반적으로 로컬 고정 추정을 만족해야 한다. 인자 $\overline{\alpha^l}$ 을 찾기 위한 효과적인 해법이, 다음의 선형 사각형 추정을 통해 Li와 Orchard에 의해 제안되었다.

$$\overline{\alpha^l} = \underset{\overline{\alpha^l}}{\operatorname{argmin}} \sum_{i=1}^n \|x_i - \sum_{t=1}^6 x_{i-t} \alpha_t\| \quad (2)$$

[0034] 여기서 x_i 는 인터레이스 이미지 안의 픽셀이고, x_{i-t} 는 픽셀 x_i 의 6개의 연결된 이웃 픽셀들이다. 수학적(2)에서 최소 사각형 문제는 다음의 행렬 형태로 쓰여질 수 있다.

$$\overline{\alpha^l} = \underset{\overline{\alpha^l}}{\operatorname{argmin}} \|\vec{x} - C\vec{\alpha}\|, \quad (3)$$

[0036] 여기서 $\vec{x} = [x_1, x_2, \dots, x_n]^T$ 는 가중치 $\overline{\alpha}$ 를 정교하게 하는데 사용되는 $M \times (M+1)$ 인터레이스 이미지 픽셀을 포함하는 벡터이다. $C = [C_0, C_1, \dots, C_n]^T$ 은 k번째 열 벡터

$C_k = [C_{0k}, C_{1k}, C_{2k}, C_{3k}, C_{4k}, C_{5k}]$ 가 x_k 주위의 가장 가까운 6개의 이웃들을 포함하는 $M \times (M+1) \times 6$ 데이터 행렬이다. 이들 알파들은 수학식(3)을 최소화해서 얻을 수 있고, 다음의 수학식(4)에 도달한다.

$$\vec{\alpha}^i = (C^T C)^{-1} C^T \vec{x} \quad (4)$$

지리적 이중성에 따라, 다음의 방정식을 얻을 수 있다.

$$\vec{\alpha}^h \cong \vec{\alpha}^i \quad (5)$$

보간된 픽셀 $y_{2i+1, j}$ 는 수학식(1)의 $\vec{\alpha}^h$ 을 식(4)의 $\vec{\alpha}^i$ 으로 치환해서 얻을 수 있다.

디인터레이스 이미지내의 손실 픽셀들은 독립적으로 보간되기 때문에, 에지 방향 적응적 인트라 필드 디인터레이싱 기법은 병렬 처리에 유리하다. 컴퓨터 연산 성능을 향상시키기 위해, 그래픽 프로세싱 유닛이 과학 및 공학 어플리케이션에 더욱더 수용되어 왔다. 따라서 이러한 형태로 본 발명의 그래픽 프로세싱 유닛 연산 활동들을 개발했고, 연구와 그 결과는 하기에 설명될 것이다.

<에지 방향 디인터레이싱의 그래픽 프로세싱 유닛 기반 구현>

에지 방향 적응적 인트라 필드 디인터레이싱 기법의 병렬 구현은 16 스트림의 다중 프로세서들(SMs)로 구성된 NVIDIA GeForce GTX 590에서 실행되며, 각각의 SM은 32개의 스트림 프로세서들(SPs)을 가진다. 따라서 단일 GTX 590에는 512개의 연산 코어들이 있다. 본 발명의 컴퓨터에 대한 명세는 테이블 1에 나타난다.

표 1

GPU	
Model	GTX90
Number of cores	512
Frequency	1.23GHz
Memory	1,536MB
Shared Memory	49,152B
Register	32,768
CPU	
Model	Intel Core i7-2600K
Frequency	3.40GHz
Memory	16.0GB

본 발명의 실험에서, CPU에서 동작하는 코드는 02로 최적화 된다.

CUDA는, 그래픽 프로세싱 유닛으로의 직접적인 프로그래밍을 제공하는 C 프로그래밍 언어의 확장인 CUDA C를 사용하여, 프로그래머들이 그래픽 프로세싱 유닛 어플리케이션을 개발할 수 있게 하는 프로그래밍 플랫폼이다. 그것의 생성은 프로그램이 데이터 레벨의 병렬화 방식으로 실행될 수 있게 한다. CUDA는 다양한 프로그래밍 방법론을 수용한다.

하나의 작업은 병렬과 직렬의 부분들로 정렬된다. CPU는 강렬한 로직을 가지는 부분을 담당하고, 그래픽 프로세싱 유닛은 독립적인 데이터에 대한 상당한 양의 연산을 가지는 부분을 담당한다. 따라서, 직렬 부분은 CPU에서 처리되고, 커널(kernel)이라 불리는 병렬 부분은 그래픽 프로세싱 유닛에서 실행된다. 아주 많은 커널의 복사본들이, CUDA 확장을 사용하는 C 프로그램에 의해 동시에 실행되는 이용 가능한 다중 프로세서들에 분배될 것이다. CUDA 코드는 그래픽 프로세싱 유닛의 전역 메모리에 데이터 전송, CUDA kernel의 실행, CPU의 메모리에 그래픽 프로세싱 유닛으로부터의 연산 결과를 전달의 3 단계로 구성된다.

본 발명의 실험에서 타겟 그래픽 프로세싱 유닛은 GTX 590이다. 도 2는 이 그래픽 프로세싱 유닛 장치의 회로도 그림을 도시한다. 각각의 다중 프로세서는 32개의 코어들을 가지고, 다른 멀티 프로세서들과 함께 병렬로 실행한다. 같은 멀티 프로세서에 있는 모든 코어들은 동시시간대에 같은 명령어를 실행한다.

- [0049] 비록 1.5GB의 전역 메모리를 가지는 타겟 그래픽 프로세싱 유닛이 더 높은 대역폭을 가지지만, CPU에 있는 DRAM 메모리와 비교하면, 멀티 프로세서 안에 공유된 메모리에 저장되는 데이터에 대한 접근은, 전역 메모리 접근이 요구하는 400~600 주기와 비교하여 밴드(band) 충돌 없이, 한 클럭 주기에 실행될 수 있다. 따라서, 자주 이용되는 데이터를, 전역 메모리 대신, 공유 메모리에 자주 유지하는 것은 좋은 선택이다. 게다가, 각각의 멀티프로세서는 한 클럭 주기에 접근될 수 있는 32K의 32비트 레지스터들을 가진다.
- [0050] 쓰레드는 CUDA 프로그래밍에서 병렬의 기본 원자 단위이다. 블록은 쓰레드들에 의해 3 단계 계층으로 구성된다.
- [0051] 같은 블록에서의 쓰레드들의 집합은, 약간의 공유 메모리를 통해 데이터를 공유하고, 그들의 실행을 동기화 하여, 서로 협력하여 동작할 수 있으며, 하나의 블록의 모든 쓰레드들은 같은 멀티프로세서 코어에서 실행된다.
- [0052] 가장 높은 레벨은 쓰레드 블록들로 구성된 그리드이다. 하나의 그리드의 모든 쓰레드 블록들은, 각각의 쓰레드 블록이 어떤 순서로든지 실행될 수 있게 하기 위해, 독립적으로 구현되며, 이러한 사실은 CUDA 프로그램이 다른 수의 프로세서들을 가지는 그래픽 프로세싱 유닛에서 실행될 수 있게 한다.
- [0053] 본 발명의 실험에서 디인터레이스 이미지로서 동작하는 기본 이미지 크기를 가지는 이미지를 사용했다. 그리고 위에서 언급한 이미지로부터 샘플링 된 (넓이) X (높이/2)의 이미지가 인터레이스 이미지로 사용되었다.
- [0054] 결국, 에지 방향 적응적 인트라 필드 디인터레이싱 기법은 인터레이스 이미지로부터 디인터레이스 이미지를 생성하기 위해 사용된다. 예를 들면, 1920x1080 크기의 이미지를 원본 이미지로 수용했다. 그리고 원본 이미지로부터 샘플링된 1920x540 크기의 이미지를 인터레이스 이미지로 사용했다.
- [0055] 결과적으로, 본 발명은 위에서 언급한 디인터레이싱 기법을 사용하여 인터레이스 이미지로부터 원본 이미지를 재생성했다.
- [0056] 본 발명의 실험에서 병렬 구조의 기본 아이디어는, 디인터레이스 이미지에서 하나의 손실 픽셀을 계산하기 위해 하나의 쓰레드를 사용하는 것이다. 그래서 1920x540 개의 쓰레드들을 가지는 커널을 설계했다. 왜냐면 1920x540 개의 손실 픽셀이 있기 때문이다. 쓰레드들은 조잡한 가변 데이터 병렬의 구현을 막고, 많은 수의 코어들에 걸쳐 어떠한 순서로도 스케줄링 될 수 있도록 독립적으로 실행된다. 이것은 CUDA 코드가 프로세서의 수를 조정할 수 있도록 한다.
- [0057] 그래픽 프로세싱 유닛 메모리는 CUDA에서 중요한 역할을 한다. CUDA에는 수많은 종류의 메모리가 있다. 도 3은 “cudaFuncCachePreferL1”의 실행이 있거나 없을 때의, L1과 공유 메모리의 메모리 할당을 보여준다. (a)는 cudaFuncCachePreferL1을 런칭한 경우이고, (b)는 cudaFuncCachePreferL1을 런칭하지 않은 경우를 도시한다.
- [0058] 전역 메모리는 CPU와 그래픽 프로세싱 유닛 사이의 통신의 주요 모드이다. 비록 텍스처 메모리와 상수 메모리가 전역 메모리인 비디오 메모리에 할당될 수 있지만, 그들은 데이터 접근을 빠르게 하기 위해서 자동으로 캐쉬된다.
- [0059] 그래픽 프로세싱 유닛의 저장 구조에서, 전역 메모리 접근 속도는 400~600 주기 사이의 지연으로 가장 낮은 속도이다. 그러나 그것의 용량은 본 발명의 알고리즘을 더욱 쉽게 구현할 수 있을 만큼 가장 크다. 초기에, 인터레이스 이미지와 중간 변수들은 전역 메모리에 저장되고, 각각의 쓰레드는 전역 메모리로부터 데이터를 가져온다. 본 발명은 원본 이미지로부터 샘플링 된 1920x540 크기의 인터레이스 이미지로부터 1920x1080 크기의 원본 이미지를 재생성하기 위한, 그래픽 프로세싱 유닛에서 동작하는 디인터레이싱의 병렬 구현을 성공적으로 실행했다. 그러나 속도 향상은, 테이블 2 에서 보이듯이, 단지 3.3배의 향상밖에 없었다.

표 2

[0060]

	Execution time(ms)	PSNR(dB)
CPU	4,898 ms	31.281
GPU	1,472 ms	31.281
Speed-up	3.3x	

[0061]

A. 공유메모리의 최적화

[0062]

에지 방향 적응적 인트라 필드 디인터레이싱 기법의 초기 그래픽 프로세싱 유닛 버전의 성능은 기대에 한참 미치지 못했다. CUDA 커널들은 CPU에서 직접적으로 접근하는 대신, 그래픽 프로세싱 유닛에서 전역 메모리에 있는

데이터를 접근했다.

[0063] 따라서, 데이터는 PCI-E x 16 하드웨어 인터페이스를 통해 상수 메모리와 텍스처 메모리와 함께, CPU 메모리에서 그래픽 프로세싱 유닛 전역 메모리로 전송되어야 했다.

[0064] 빠른 메모리 접근을 위해, 공유 메모리나 레지스트들이 활용 가능하다. 전역 메모리의 접근은 400~600 클럭 주기를 요구한다. 전역 메모리 접근의 높은 지연은 병렬 컴퓨팅의 병목 현상 중의 하나이다. 이 지연을 없애기 위해, 자주 이용되는 데이터는, 빠른 접근을 위해, 전역 메모리로부터 공유 메모리로 전송된다. 공유 메모리는, 같은 쓰레드 블록에서, 쓰레드들 간의 통신을 허용한다.

[0065] 본 발명의 디인테레이싱 알고리즘에서 자주 이용되는 데이터를 빠른 접근을 위해 전역 메모리 대신 공유 메모리에 저장했다. 이는 공유 메모리 접근의 대역폭이 거의 레지스트만큼 빠르기 때문이다. 하지만 공유 메모리는 그것의 제한된 공간과 메모리 뱅크 충돌 때문에 효율적으로 사용되어야 한다.

[0066] 공유 메모리에 저장된 데이터는 쓰레드 인덱스에 의해 각 쓰레드에서 접근될 수 있다. 공유 메모리는 32개의 뱅크들로 나누어진다. 이들 쓰레드들이 다른 뱅크들로부터 동시에 데이터를 접근할 수 있게 하기 위해, 하나의 블록 내에 있는 모든 쓰레드들에 대해 동기화가 필수이며, 그로 인해 그들은 데이터에 접근하기 위해 공유 메모리를 사용할 때, 같은 단계들을 실행할 수 있다.

[0067] GTX 590에는 각각의 SM에 49152 바이트의 공유 메모리가 있다. 공유 메모리의 크기를 고려하여, 본 발명은 하나의 1920x1080 이미지를 처리하기 위해 4x4 크기의 블록을 사용했다. 이 경우에, 얻어진 속도 향상은 테이블3에서 보여주듯이 10.0배였다.

표 3

[0068]

	Execution time(ms)	PSNR(dB)
CPU	4,898 ms	31.281
GPU	489 ms	31.281
Speed-up	10.0x	

[0069] **B. 레지스터 메모리를 이용한 더 향상된 최적화**

[0070] 레지스터 메모리는 빠른 메모리 공간이며, 특정 쓰레드들에 속해있으며, 자동으로 생성된 일부 변수들을 저장하는데 사용된다. 그것의 기본 단위는 레지스터 파일이며, 각각의 레지스터 파일의 크기는 32비트이다. 일반적으로, 레지스터의 접근은, 명령 당 추가적인 클럭 주기를 요구하지 않는다. 공유 메모리를 사용하는 대신, 우리는 또한 레지스터의 사용을 또한 실험했으며, 테이블 4에서 보여주듯이 괜찮은 성능 향상을 얻었다.

표 4

[0071]

	Execution time(ms)	PSNR(dB)
CPU	4,898 ms	31.281
GPU	119 ms	31.281
Speed-up	41.2x	

[0072] **C. 공유 메모리 보다 L1 캐쉬를 사용하여 얻은 더 향상된 최적화**

[0073] 레지스터의 사용 절차에서, 레지스터의 개수 제한 때문에, 일부 데이터는 로컬 메모리에 저장될 것이다. 로컬 메모리의 접근은 전역 메모리의 접근처럼 매우 비용이 크다. 본 발명의 실험에서 그래픽 프로세싱 유닛은 Fermi 아키텍처를 사용하여 만들어졌다. 각각의 SM은 64KB 크기의 설정 가능한 칩내 캐쉬를 가지며, 이는 48KB의 공유 캐쉬와 16KB의 L1 캐쉬로 설정될 수 있다. 또한 도 4에서 보여주듯이, 공유 캐쉬를 16KB로, L1 캐쉬를 48KB로 설정할 수 있다.

[0074] L1 캐쉬는 레지스터 오버플로우, 스택 연산, 전역 DL/ST 등을 다루기 위해 사용될 수 있다.

[0075] Fermi 768KB 통합 L2 캐쉬는 모든 저장 연산들을 지원할 수 있다. L2 캐쉬는 그래픽 프로세싱 유닛으로의 효과적이고 높은 성능의 데이터 공유를 제공한다. 본 발명의 보간 알고리즘에서, 내부 쓰레드 통신의 필요성은 없다. 따라서, 우리는 또한 이미지 데이터의 저장을 위해 레지스터들을 사용하는 것을 실험했다. 레지스터 낭비

를 경감하기 위해, “cudaFuncCachePerferL1” 명령은 우리의 코드에서 실행되었으며, 그것은 공유 메모리의 사용보다 L1 캐쉬의 사용이 더 많다는 것을 의미한다. L1 캐쉬 명령의 사용이, 사용하지 않았을 때보다 더 나은 컴퓨팅 성능을 보여준다는 것을 발견하였다. 그 결과는 테이블5에서 볼 수 있다.

표 5

	Execution time(ms)	PSNR(dB)
CPU	4,898 ms	31.281
GPU	107 ms	31.281
Speed-up	45.8x	

공유 메모리 대신 L1 캐쉬를 더 사용하자는 이 제안은 디인터레이싱 절차의 속도 향상을 도와준다.

D. 비동기 데이터 전송을 이용하여 더욱 속도 향상

그래픽 프로세싱 유닛은 그것의 엄청나게 많은 코어 때문에, 상당한 컴퓨팅 성능을 가진다. 하지만, 그래픽 프로세싱 유닛에서 다루지는 데이터는 호스트 장치에서 그래픽 프로세싱 유닛 장치로 전달되어야 하고, 그 결과는 다시 호스트 장치의 메모리도 돌려줘야 한다.

호스트 장치와 그래픽 프로세싱 유닛 장치간의 데이터 전송 절차는 상당한 양의 시간을 필요로 한다. 호스트 장치 데이터 전송 시간을 줄이고, 속도 향상을 얻기 위해, 예지 방향 적응적 인트라 필드 디인터레이싱 기법을 위해 비동기 데이터 전송(ADT)을 더 구현했다. ADT를 사용한다는 것은 2개 혹은 더 많은 CUDA 스트림이 더 필요하다는 것을 의미한다. 본 발명은 하나의 스트림이 연산을 위해 커널을 실행할 수 있도록 허용하는 동안, 두 번째 스트림이 호스트 장치와 그래픽 프로세싱 유닛 장치사이의 데이터 전송을 실행하도록 허용할 수 있다. CUDA 스트림은 특정 순서에서 실행되는 그래픽 프로세싱 유닛 연산들의 큐로 언급될 수 있다. 커널 실행, 메모리 복사, 이벤트 시작/종료를 포함하는 큐는 CUDA 스트림에 추가될 수 있다. 각각의 스트림은 그래픽 프로세싱 유닛에서의 하나의 작업으로 여겨질 수 있으며, 이들 작업들은 병렬로 실행될 수 있다. 일반적으로 다중 스트림들은 ADT를 인식하기 위해 사용될 수 있고, 우리는 본 발명의 디인터레이싱 기법에서 두 개의 스트림 ADT를 사용한다.

사실, 스트림의 개념은 다소 오해의 소지가 없지 않다. CUDA는 메모리 복사와 커널 실행을 다루는 독립적인 엔진을 가지며, 따라서 우리는 스트림에 추가되는 연산들의 큐에서의 순서는, CUDA 드라이버가 실행을 스케줄링하는 방식에 영향을 준다는 사실을 명심해야 한다. 두 번째 스트림에 있는 블록은, 특정 스트림의 모든 연산들이 스케줄링 되면, 쉽게 실행될 수 있다. 하지만 우리가, 다른 스트림들의 연산들을 추가하기 위한 번갈아 실행하는 방식을 수용하면, 문제는 효과적으로 경감될 수 있다.

우리의 디인터레이싱 기법에 적용된 두 개의 스트림 ADT는 도 5에 도시된다.

도 5로부터, 본 발명은 하나의 스트림이 커널 실행을 다룰 때, 다른 스트림이 호스트 메모리와 그래픽 프로세싱 유닛 메모리 간의 데이터 전송을 실행한다는 것을 볼 수 있다. 이 속도 향상은 테이블 6에서 보듯이 향상되었다.

표 6

	Execution time(ms)	PSNR(dB)
CPU	4,898 ms	31.281
GPU	102 ms	31.281
Speed-up	48.0x	

E. 예지 방향 디인터레이싱의 다중 그래픽 프로세싱 유닛 구현

GTX 590은 두 개의 그래픽 프로세싱 유닛을 가지고 있다. 두 그래픽 프로세싱 유닛이 한 번에 사용되면, 실행 시간은 더욱 감소한다. 각각의 그래픽 프로세싱 유닛은 이미지의 반을 처리한다. 제안된 다중 그래픽 프로세싱 유닛은, 다중 연산 장치들을 가지는 단일 노드 데스크탑 시스템들을 나타낸다. 기본 아이디어는 주소 포인터로 다른 그래픽 프로세싱 유닛을 구별하는 것이다. 이는 테이블 7에서 보여주듯이 최고의 성능을 얻을 수 있다.

표 7

[0087]

	Execution time(ms)	PSNR
CPU	4,898ms	31.281
GPU	51.8ms	31.281
Speedup	94.6x	

[0088]

그래픽 프로세싱 유닛 기반 디인터레이싱 기법의 컴퓨팅 성능에서의 스레드 블록 크기의 효과를 알기 위해, 우리는 하나의 1920x1080 이미지를 처리할 때, 다른 블록 크기에 대해 성능 향상을 시험했다. 결과는 도 6에서 볼 수 있으며, 블록 크기가 128일 때 최고의 성능을 보여준다.

[0089]

그래픽 프로세싱 유닛에서의 에지 방향 적응적 인트라 필드 디인터레이싱 기법을 입증하기 위해, 도 7(a)~9(a)까지의 약간의 비디오 크기 이미지들을 원본 이미지로 채택했다. 그리고 도 7(b)~9(b)까지의 인터레이스된 이미지들을 얻기 위해 그들을 샘플링 했다.

[0090]

인터레이스 이미지들을 그래픽 프로세싱 유닛 기반 에지 방향 적응적 인트라 필드 디인터레이싱 기법 코드로의 입력으로 사용하면서, 도 7(c)~9(c)까지의 디인터레이스 이미지들을 얻었고, 이들은 원본 이미지와 비슷했다. 도 7(d)~9(d)는 CPU의 직렬 코드로 디인터레이스된 이미지들을 보여준다.

[0091]

그래픽 프로세싱 유닛들 사이의 성능을 비교하기 위해, 에지 방향 적응적 인트라 필드 디인터레이싱 기법의 그래픽 프로세싱 유닛 기반 병렬 버전이 다중 NVIDIA 그래픽 프로세싱 유닛들에 동작했다. 특히, 병렬 버전은 512 코어들을 가지는 GTX590 기반 Fermi, 480 코어들을 가지는 GTX480 기반 Fermi, 192 코어들을 가지는 GTX260 기반 Fermi에서 실행되었다. 도 10에 도시된 바와 같이, 본 발명은 위의 각각 다른 그래픽 프로세싱 유닛에 대해, 비디오 크기 이미지들에 대한 다른 성능을 얻었다.

[0092]

<결론>

[0093]

에지 방향 적응적 인트라 필드 디인터레이싱 기법이 좋은 가시 품질을 달성하는 반면, 에지 주변의 각각의 보간된 픽셀에 대해 몇몇의 행렬 연산들이 실행되어야 하는 것은 상대적으로 느린 성능을 야기시킨다. 이 발명에서 NVIDIA GeForce GTX 590 그래픽 프로세싱 유닛에서의 에지 방향 적응적 인트라 필드 디인터레이싱 기법을 구현했다. 데이터 전송의 통신 대역폭이, 그래픽 프로세싱 유닛에서의 CUDA 코드의 실행 성능을 제한하는 주요 문제 중 하나였다. 이 문제를 해결하기 위해, 컴퓨팅 성능을 개선하기 위한 몇 가지 노력을 했다. 예를 들면, 레지스터와 공유 메모리의 사용이 고려되었고, 공유 메모리보다 더 많은 메모리로 L1 캐시를 이용했고, 블록 당 스레드들의 효과가 연구되었다.

[0094]

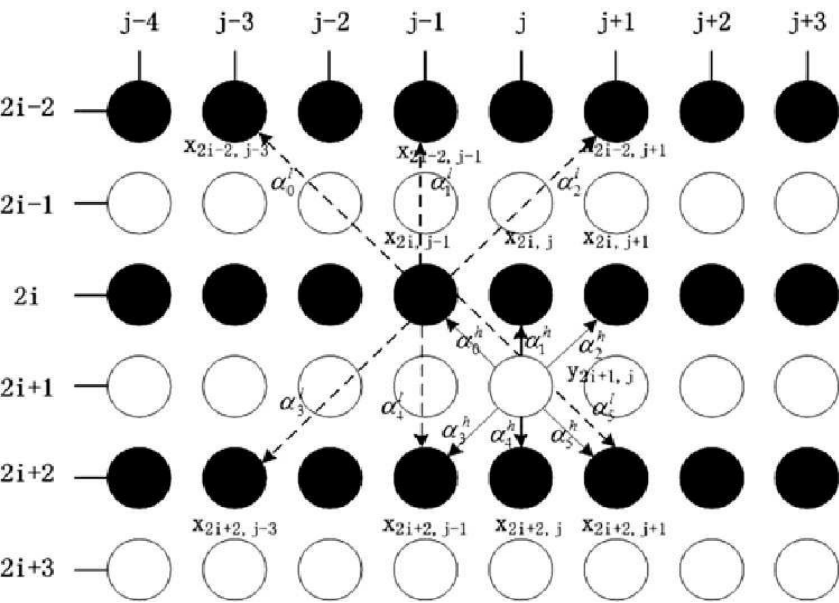
1920x540의 인터레이스 이미지를 1920x1080 디인터레이스 이미지로 디인터레이싱 하기 위한 CUDA 구현에서 3.40GHz 인텔 코어 i7 호스트 컴퓨터의 단일 스레드 CPU와 비교하여, I/O를 포함하여 94.6배의 성능 향상을 얻을 수 있었다.

[0095]

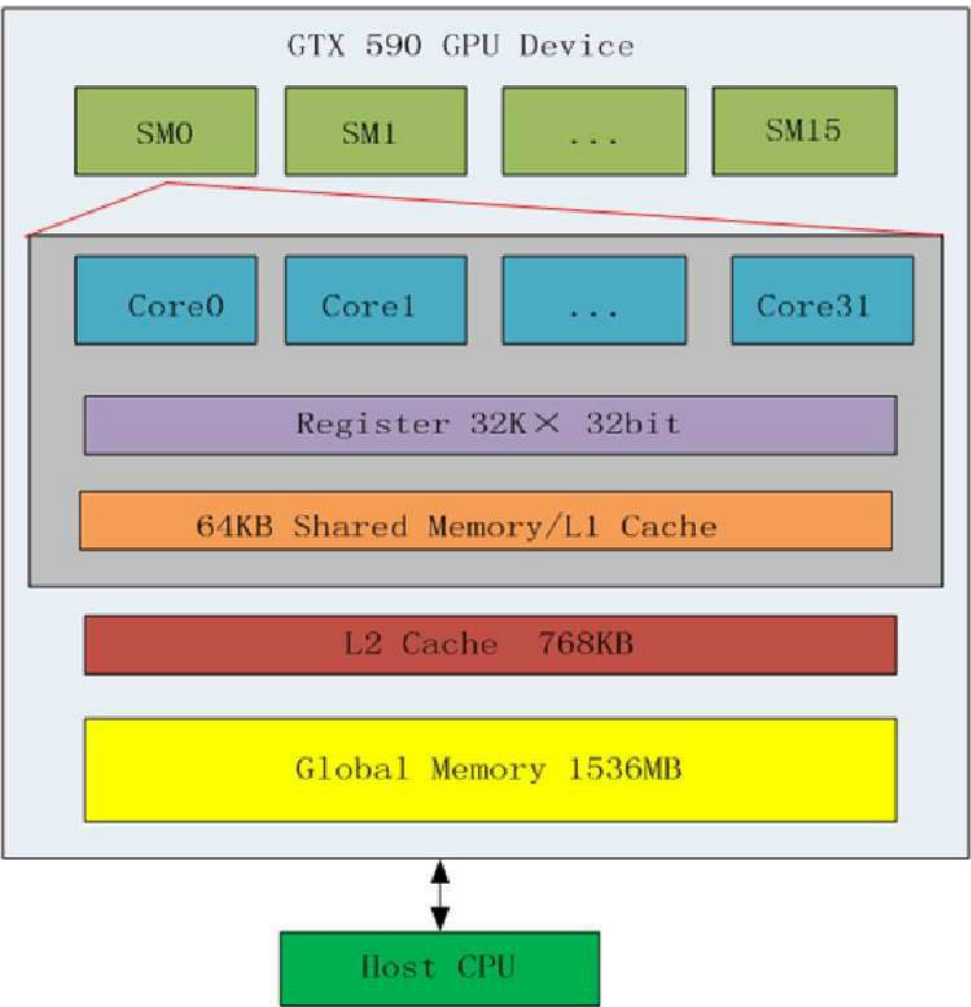
이상에서 설명된 본 발명의 에지 방향 적응적 인트라 필드 디인터레이싱 기법을 이용한 그래픽 프로세싱 유닛의 병렬 구현 방법의 실시 예는 예시적인 것에 불과하며, 본 발명이 속한 기술분야의 통상의 지식을 가진 자라면 이로부터 다양한 변형 및 균등한 타 실시예가 가능하다는 점을 잘 알 수 있을 것이다. 그러므로 본 발명은 상기의 상세한 설명에서 언급되는 형태로만 한정되는 것은 아님을 잘 이해할 수 있을 것이다. 따라서 본 발명의 진정한 기술적 보호 범위는 첨부된 특허청구범위의 기술적 사상에 의해 정해져야 할 것이다. 또한, 본 발명은 첨부된 청구범위에 의해 정의되는 본 발명의 정신과 그 범위 내에 있는 모든 변형물과 균등물 및 대체물을 포함하는 것으로 이해되어야 한다.

도면

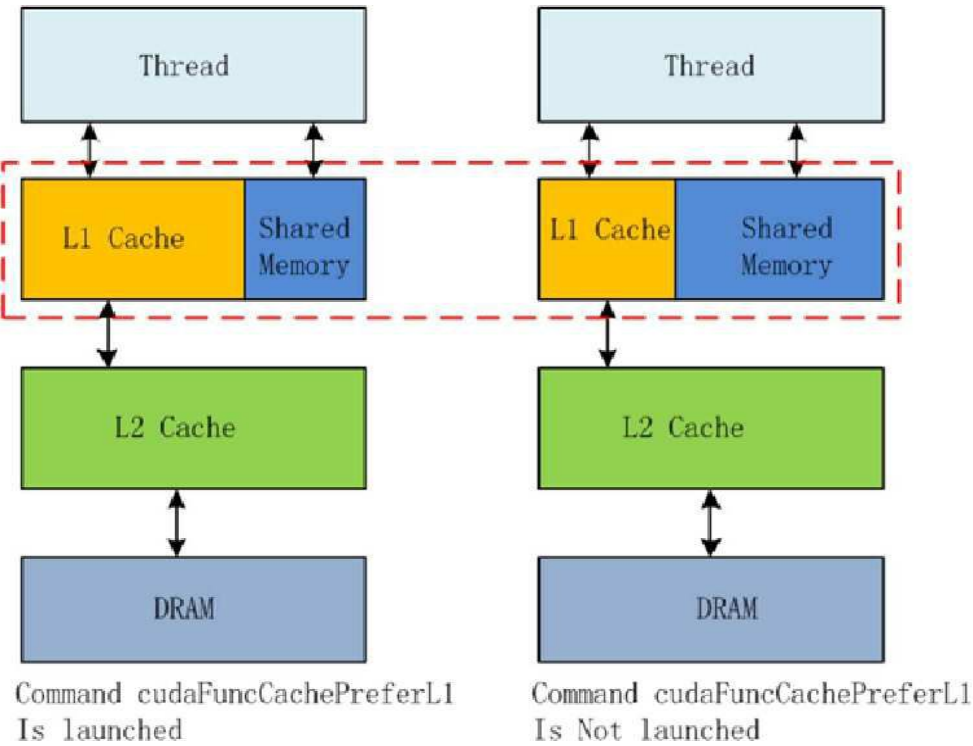
도면1



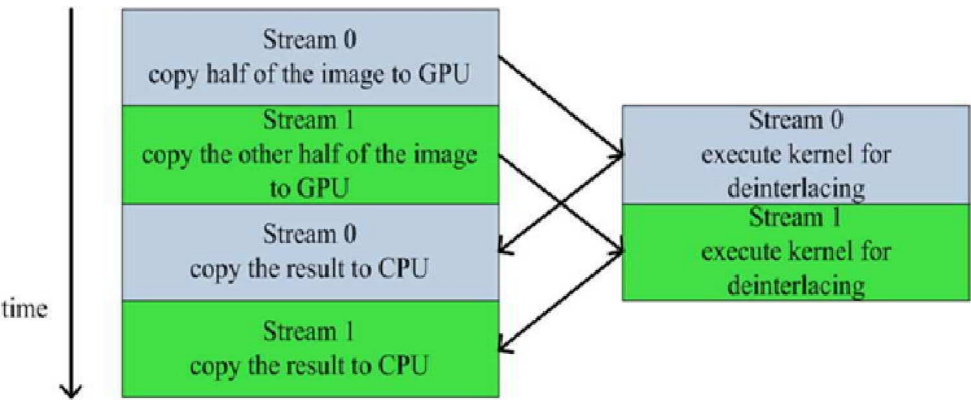
도면2



도면3



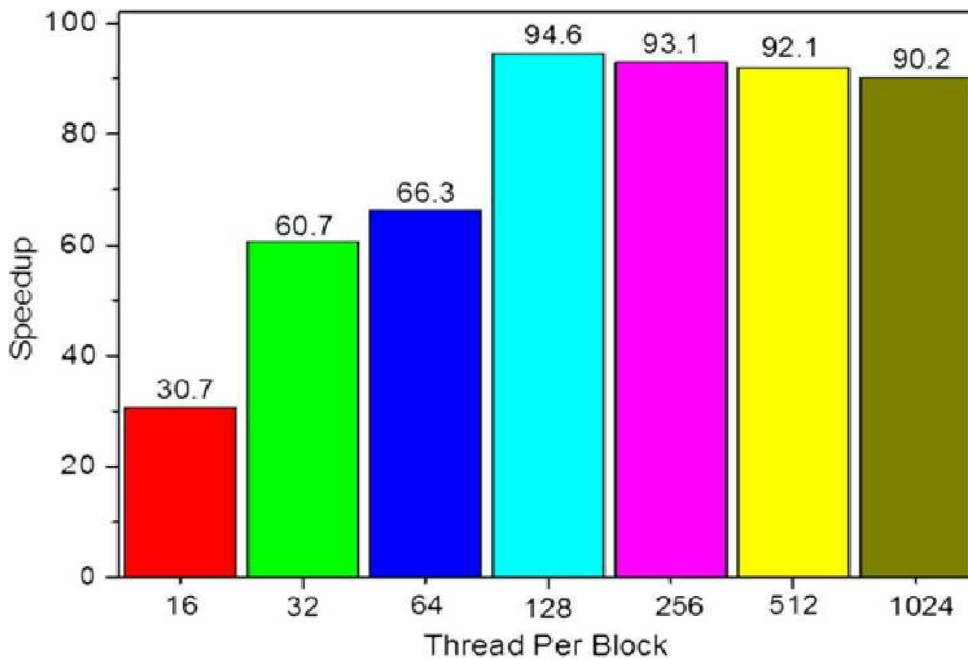
도면4



도면5

```
//copy the data from the CPU to the GPU
for(int j=0; j<GPUs; j++){
    cudaSetDevice(j);//choice device
    //d_data is a device pointer;var is a host pointer
    cudaMemcpyAsync(d_data[0][j],var[0][j],size*sizeof(float),cudaMemcpyHostToDevice,stream[0][j]);
    cudaMemcpyAsync(d_data[1][j],var[1][j],size*sizeof(float),cudaMemcpyHostToDevice,stream[1][j]);
}
//invoke the kernel
for(int j=0;j<GPUs;j++){
    cudaSetDevice(j);
    kernel_h<<<grid,threads,0,stream[0][j]>>>(d_data[0][j],M);//M is the width of image
    kernel_h<<<grid,threads,0,stream[1][j]>>>(d_data[1][j],M);
}
for(int j=0;j<GPUs;j++){
    cudaSetDevice(j);
    cudaStreamSynchronize(stream[0][j]);
    cudaStreamSynchronize(stream[1][j]);
}
//copy the data from the GPU to the CPU
for(int j=0;j<GPUs;j++){
    cudaSetDevice(j);
    cudaMemcpyAsync(h_data+(6+2*j*size1)*M,d_data[0][j]+6*M,size1*M*sizeof(float),
    cudaMemcpyDeviceToHost,stream[0][j]);//the array h_data is the data of image
    cudaMemcpyAsync(h_data+(6+(2*j+1)*size1)*M,d_data[1][j]+6*M,size1*M*sizeof(float),
    cudaMemcpyDeviceToHost,stream[1][j]);
}
```

도면6



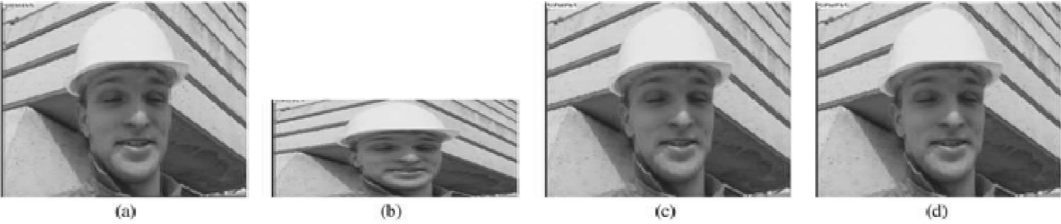
도면7



도면8



도면9



도면10

