Journal of The Korea Society of Computer and Information
Vol. 19, No. 4, April 2014

www.ksci.re.kr
http://dx.doi.org/10.9708/jksci.2014.19.4.009

GPU를 이용한 선형 스크래치 탐지와 복원 알고리즘의 설계

이 준 구*, 심 세 용*, 유 병 문**, 황 두 성***

Design of Line Scratch Detection and Restoration Algorithm using GPU

Joon-Goo Lee*, She-Yong Shim*, Byoung-Moon You**, Doo-Sung Hwang***

요 약

본 논문은 화소 데이터의 비교를 이용한 단일 프레임 또는 연속 프레임에 나타나는 선형 스크래치를 탐지하여 복원하는 알고리즘을 제안하였다. 스크래치 탐지와 복원 방법은 프레임 간 많은 비교 연산 시간을 필요로 하며 병렬처리 가능성이 높다. 제안하는 스크래치 탐지와 복원 방법은 빠른 처리를 위해 GPU에서 수행할 수 있도록 병렬 설계 하였다. 제안하는 알고리즘은 국가 기록원 디지털화 영상에 대해 순차처리와 병렬처리의 성능 테스트를 수행하였다. 실험에서 연속한 스크래치를 고려하는 경우의 탐지율은 단일 프레임만 고려하는 방법보다 20% 이상 성능이 향상되었다. GPU 기반 알고리즘의 탐지율과 복원율은 CPU 기반의 알고리즘과 유사하였으나 50배 이상의 연산 속도가 향상되었다.

▸ Keywords : 스크래치 탐지, 스크래치 복원, 병렬처리, GPU

Abstract

This paper proposes a linear scratch detection and restoration algorithm using pixel data comparison in a single frame or consecutive frames. There exists a high parallelism in that a scratch detection and restoration algorithm needs a large amount of comparison operations. The proposed scratch detection and restoration algorithm is designed with a GPU for fast computation. We test the proposed algorithm in sequential and parallel processing with the set of digital videos in National Archive of Korea. In the experiments, the scratch detection rate of consecutive frames is as fast as about 20% for that of a single frame. The detection and restoration rates of a

•제1저자 : 이준구 •교신저자 : 이준구
•투고일 : 2013. 11. 15, 심사일 : 2013. 12. 10, 게재확정일 : 2014. 2. 28.
* 단국대학교 컴퓨터과학과(Dept. of Computer Science, Dankook University)
** (주)엘앤와이비젼(L&Y Vision Technologies, Inc)
*** 단국대학교 운동의과학과(Dept. of Kinesiologic Medical Science & Computer Science, Dankook University)
※ 이 논문은 행정안전부 국가기록원 재원으로 2012년 기록보존 기술 연구개발사업의 지원을 받아 수행됨.

GPU-based algorithm are similar to those of a CPU-based algorithm, but the parallel implementation speeds up to about 50 times.

▸ Keywords : scratch detection, scratch restoration, parallel processing, GPU

I. 서 론

디지털 기술이 발전하며 효율적인 멀티미디어 서비스를 위해 과거에 만들어진 아날로그 영상들을 디지털화 하고 있다. 하지만 오래된 아날로그 필름의 보관, 영사 또는 복사 등의 과정에서 원본 영상에 훼손이 발생함에 따라 디지털화된 영상에도 훼손이 나타나며, 특히 선형 스크래치는 많이 발생하는 훼손중의 하나이다[1]. 기록물 보존은 최대한 원본에 가까운 상태로 유지되어야 하기 때문에 디지털화 영상의 훼손에 대한 고속 자동 검출 및 복원 기술이 요구된다.

선형 스크래치의 탐지 및 복원은 대용량의 데이터 처리에 대해 높은 연산량이 요구되며, 시간적, 공간적으로 많은 비용이 소요된다. 이러한 문제는 병렬 처리를 이용해 해결할 수 있으나, CPU(Central Processing Unit)는 분기문이 복잡하고 수용 가능한 스레드의 수가 적은 관계로 높은 병렬성을 효과적으로 처리하기에 한계가 있다. 선형 스크래치의 탐지는 연속된 프레임 간 화소 비교 연산 수행이 필요하며, 프레임 간 화소 정보의 단순 비교 연산은 대량의 코어를 탑재한 GPU(Graphics Processing Unit)를 활용하면 효과적인 병렬 처리가 가능하다[2].

본 논문에서는 스크래치 탐지와 복원 알고리즘을 제안하고, CUDA를 이용한 GPU 기반의 병렬 설계를 통해 성능 향상을 보인다. 2장에서는 관련 연구와 CUDA를 소개하고, 3장에서는 제안하는 알고리즘과 병렬 설계 방법을 제안한다. 4장에서는 제안된 알고리즘들의 성능을 국가기록원 소장 디지털화 영상에 적용하여 비교하며, 5장에서는 제안하는 방법의 문제점과 개선 방향에 대해 논한다.

II. 관련 연구

1. 스크래치 탐지 및 복원

스크래치는 주변 화소에 비해 급격하게 밝거나 어두운 화소가 세로로 길게 연결된 형태로 나타나며, 다양한 길이와 너비, 유동적인 위치, 그리고 연속한 프레임에 걸쳐 나타날 수 있다. 스크래치의 종류는 Static, Moving, Principal, Secondary, Alone, Not-alone, Negative, Positive로 분류하며, 탐지 와 복원을 위한 다양한 연구가 수행되었다[3].

신경망 기반의 텍스처 분류기와 형태학적 필터링을 사용하여 스크래치를 탐지한 후, 양선형 보간법(bilinear interpolation)을 사용해 복원하는 방법이 소개되었다[3]. Kokaram은 횡단면의 휘도 차이에 의한 선의 윤곽(line profile) 모델을 통해 스크래치를 탐지한 후, 2차원 자동 회귀(2D Autoregressive)를 이용해 스크래치를 복원하였다[4]. Bruni 등은 Kokaram's 모델을 이용해 스크래치를 탐지하며, 완전한 자동화를 위해 웨버의 법칙(Weber's law)에 따른 적응적 임계값을 사용하였다[5]. Gullu 등은 Bruni 등의 방법을 이용해 프레임 단위로 스크래치 후보를 검출한 후 인접한 일련의 프레임에서 스크래치 위치가 일관된 것들만을 탐지하였으며, 위치 정보를 이용하여 연속된 프레임 간의 인접한 위치의 화소 정보들을 이용해 복원하였다[6].

기존의 방법들은 수직 방향의 평균 또는 분산을 구하여 일차원 신호를 형성하기 때문에 스크래치의 위치는 탐지할 수 있지만 정확한 길이는 탐지할 수 없었다. 또한 프레임의 세로축 길이에 대해 스크래치의 길이가 차지하는 비율을 이용하기 때문에 연속한 프레임에 걸쳐 나타나는 스크래치는 탐지할 수 없었다[7]. Gullu 등의 방법 역시 기본적인 스크래치 후보 검출은 단일 프레임을 단위로 하기 때문에 프레임에 연속적으로 나타나는 스크래치는 탐지하지 못한다. 따라서 스크래치의 정확한 길이, 위치, 프레임에 연속적인 스크래치를 탐지할 수 있는 방법이 보다 일반적이다.

2. CUDA

GPU는 수 백개의 부동 소수점 계산 유닛과 독립된 메모리를 가지고 있다. 컴퓨터 그래픽스를 위한 GPU를 이용해 CPU가 처리하던 응용 프로그램들의 계산을 지원하는 범용 GPU(GPGPU, General-Purpose computing on GPU) 기술이 활발히 사용되고 있으며, 연산 능력이 뛰어나 활용 가

능성이 매우 높은 기술이다. CUDA(Compute Unified Device Architecture) 병렬 컴퓨팅 아키텍처는 CPU(또는 host)와 GPU(또는 device)의 co-processing 처리 환경을 제공하며, GPU에서 수행하는 병렬 처리 알고리즘을 C 프로그래밍 언어를 사용해 작성할 수 있는 구조이다.

CUDA는 여러 개의 SM(Streaming Multiprocessor)으로 구성되어 있고, 하나의 SM은 그림 1과 같이 여러개의 SP(Streaming Processor)로 구성되는 계층적 구조이다. 워프(warp)는 스레드의 스케줄링을 담당하는 단위로, 하나의 워프가 32개의 스레드를 관리한다. GTX580의 경우 32개의 SP가 모여 하나의 SM을 이루며, 각 SM은 최대 48개의 워프를 지원하므로 24,576개의 스레드를 동시에 생성할 수 있다. CUDA의 메모리 구조는 그림 2와 같이 구성되어 있다. 레지스터는 각 스레드의 변수를 저장하며 가장 빠른 접근 속도를 갖는다. 공유 메모리(Shared Memory)는 블록 내 스레드 들이 데이터를 공유할 수 있으며 캐시 기능이 있다. 전역 메모리(Global Memory)는 접근 속도는 느리지만 대량의 데이터를 저장할 수 있으며 host와 통신이 가능하다.

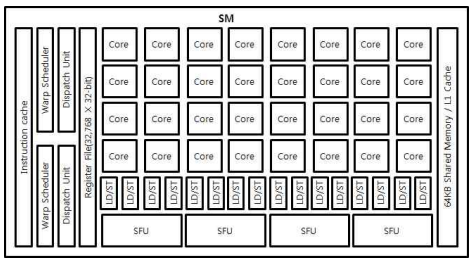

그림 1. CUDA의 SM 구조
Fig. 1. SM structure of CUDA

GPU를 이용한 병렬처리는 SIMT(Single Instruction, Multiple Threads) 방식의 처리를 이용한다. 반복적으로 나타나는 연산을 수 만개의 스레드로 분할하여 순차성이 없는 대량의 데이터에 대해 적용함으로써 GPU의 수 백개의 코어(Core)가 대량의 데이터 연산을 빠르게 처리할 수 있다. 대용량 행렬 곱셈과 Sum Reduction에서는 공유 메모리를 사용하여 대용량 데이터의 접근 속도를 줄여 효율적인 병렬처리를 보였다[8-10]. HP는 가변 데이터 인쇄 시스템 고속 결함 검출을 위해 SSIM (Structural Similarity Information Measure), DSIM (structural Dis-Similarity Index Measure) 연산들을 GPU의 유닛에 의존하여 계산했으며, 2차원 메모리 접근 패턴을 최적화하기 위하여 텍스처 메모리를 이용해 효율적인 병렬 처리를 시도하였다[11]. CUDA의 기술이 보편화되면서 많은 학술 논문과 함께 다양한 분야의 응용 프로그램들이 속도 향상의 결과를 보이고 있다[12].

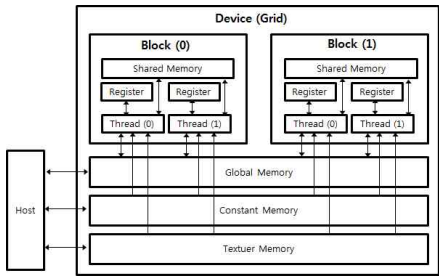

그림 2. CUDA의 계층적 메모리 구조
Fig. 2. Hierarchical memory architecture of CUDA

III. 본 론

스크래치 탐지 및 복원 방법은 연속된 프레임으로 구성된 비디오 $F = (F_0, F_1, F_2, \ldots F_{n-1})$에 대해 스크래치를 탐지하여 복원된 비디오 $R = (R_0, R_1, R_2, \ldots R_{n-1})$을 출력한다. F는 N개의 프레임으로 구성된 비디오 데이터, F_n은 크기가 $X \times Y$인 2차원 행렬, $F_n(x,y)$는 n 번째 프레임의 좌표 (x,y)의 화소 밝기 값이다. 제안하는 방법은 F_n와 연속한 F_{n+1}을 흑백으로 변환한 후, 작은 크기의 스크래치 세그먼트를 단위로 하여 검출한다. 두께별 스크래치 세그먼트를 각각 검출하며, 스크래치 세그먼트가 표시된 연속한 프레임을 상하로 연결하여 가상 프레임을 생성 후 연속한 프레임에 걸쳐 나타나는 선형 스크래치 까지 탐지한다. 탐지된 스크래치는 인접 화소들의 정보를 이용해 복원시킨다(그림 3). 제안하는 알고리즘은 GPU 기반의 병렬 설계를 통해 고속화하며, 효율적인 병렬화를 위해 CPU와 GPU 간의 데이터 이동을 최소화하고 공유 메모리를 사용한다.

그림 3. 스크래치 탐지 과정
Fig. 3. Scratch Detection Process

1. 스크래치 탐지

스크래치를 탐지하기 위해서는 먼저 프레임 내에서 급격하게 밝거나 어두운 스크래치 화소들을 검출하며, 스크래치의 두께가 다양하게 나타나기 때문에 두께별 스크래치 화소를 각각 검출한다. F_n의 스크래치 화소는 기준 화소와 인접한 좌우 화소들 간의 비교를 통해 검출하며, 이를 계산하는 $D_n(x,y)$는 다음과 같다.

$$D_n(x,y)=\sum_{i=\Delta}1(F_n(x,y)-F_n(x+i,y)-\theta_1) \cdots (1)$$

여기서, Δ가 x축으로 인접한 화소의 거리를 나타낼 때, 얇은 스크래치는 $F_n(x,y)$를 기준 화소로 하여 $\Delta=\{-2, -1, 1, 2\}$인 주변 화소와 비교하고, 두꺼운 스크래치는 $F_n(x,y)$와 $F_n(x+1,y)$를 기준으로 하여 $F_n(x,y)$는 $\Delta=\{-2, -1\}$들과 비교하고, $F_n(x+1,y)$는 $\Delta=\{2, 3\}$들과 비교한다. θ_1은 화소 값의 차이가 스크래치를 나타내는지 판단하는 임계값이다. $x)0$이면 $1(x)=1$, 그 외는 $1(x)=0$이다. 만약 $D_n(x,y)$의 값이 Δ의 요소 개수 $|\Delta|$와 같으면 해당 화소를 스크래치 화소로 설정한다.

스크래치는 프레임 내에서 연속적이지 않을 수 있기 때문에 먼저 작은 스크래치 세그먼트 단위로 검출한다. $D_n(x,y)$가 스크래치 화소이면서, 세로축으로 인접한 화소들도 스크래치 화소인 경우, 해당 영역을 스크래치 세그먼트로 설정한다. $D_n(x,y)$를 중심으로 하여 스크래치 세그먼트를 계산하는 $S_n(x,y)$는 다음과 같다.

$$S_n(x,y)=\sum_{j=y-\tau/2}^{y+\tau/2}1(D_n(x,j)-|\Delta|) \cdots (2)$$

여기서, τ는 세그먼트의 길이를 나타내며, $S_n(x,y)$가 τ의 70% 이상이면 $F_n(x,y-\tau/2)$부터 $F_n(x,y+\tau/2)$ 영역을 스크래치 세그먼트로 설정한다.

스크래치는 단일 프레임뿐만 아니라 연속한 프레임에 걸쳐 나타날 수 있기 때문에 그림 4와 같이 스크래치 세그먼트가 표시된 F_n과 F_{n+1}을 상하로 연결하여 가상 프레임을 생성한다. 생성된 가상 프레임에 마스크를 슬라이딩 윈도우 방식으로 적용하여 단일 프레임 내에 나타나는 스크래치뿐만 아니라 연속한 프레임에 걸쳐 나타나는 스크래치를 탐지한다. 이때, 마스크의 크기는 프레임 한 장의 크기와 같고, 가상 프레임의 위에서 아래 방향으로 적용하며, 마스크 내의 각 세로축 스크래치 세그먼트의 길이를 조사하여 그 길이가 프레임의 세로 길이에 일정 비율 이상일 경우 실제 스크래치로 탐지한다.

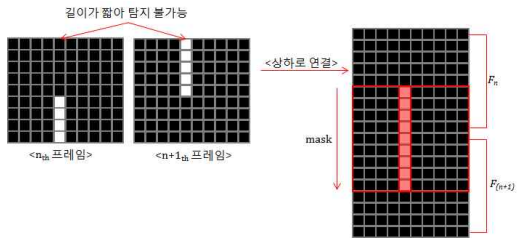

그림 4. 실제 스크래치 탐지 방법
Fig. 4. Method of real scratch detection

2. 스크래치 복원

탐지된 스크래치는 컬러 공간 별 각 스크래치 화소들의 정보를 이용하여 복원한다. 얇은 스크래치는 기준 화소가 $F_n(x,y)$인 경우 기준 화소를 제외한 $\Delta=\{-2, -1, 1, 2\}$이 인접 화소가 되며, 두꺼운 스크래치는 기준 화소가 $F_n(x,y)$와 $F_n(x+1,y)$인 경우 기준 화소를 제외한 $\Delta=\{-2, -1, 2, 3\}$이 인접 화소가 된다. 이때, $F_n(x,y)$의 화소를 복원하는 경우 좌측의 화소들에 가중치를 높이 부여하여 평균을 계산하며, $F_n(x+1,y)$의 화소를 복원하는 경우 우측의 화소들에 가중치를 높이 부여하여 평균을 계산한다. 인접 화소들 중 평균값과 가장 유사한 화소를 선택하여 기준이 되는 얇은 스크래치 화소와 굵은 스크래치 화소를 복원 한다. 각 컬러 공간의 인접 화소의 평균과 유사한 화소를 이용하여 스크래치를 복원하는 $R_n(x,y,c)$는 다음과 같다.

$$R_n(x,y,c)=\min_{i\in\Delta}(|F_n(x+i,y,c)-E(F_n(x,y,c))|) \cdots (3)$$

여기서, $E(F_n(x,y,c))=avg_{i\in\Delta}(F_n(x+i,y,c))$이고, c는 컬러 공간을 나타낸다.

3. GPU 기반 병렬 설계

CUDA의 병렬 설계는 순차성이 없는 SIMT 구조가 최적화 되어 있으므로, 제안하는 방법의 얇은 스크래치 세그먼트, 굵은 스크래치 세그먼트, 선형 스크래치 탐지 모듈을 SIMT 식으로 구조화 시킨다. host의 메모리와 device의 전역 메모리 간의 전송은 상당한 시간을 소모하기 때문에 이들 간의 불필요한 메모리 전송을 최소화 한다. device의 전역 메모리는 host와 통신하며 대량의 데이터를 저장할 수 있지만 GPU 내에서는 공유 메모리에 비해 상당히 느리기 때문에 전역 메모리의 데이터를 공유 메모리로 로드하여 데이터 접근 시간을

줄임으로써 효율적인 병렬 수행을 할 수 있다[13].

스크래치 화소를 탐지하는 식 (1)은 얇은 스크래치 화소의 경우 하나의 스레드가 하나의 화소를 기준으로 하고, 두꺼운 스크래치 화소의 경우 하나의 스레드가 두 개의 화소를 기준으로 하여 계산 하도록 한다. X와 Y가 각각 프레임의 가로와 세로의 화소 수인 경우 $X \times Y$개의 스레드를 구성하여 병렬로 처리하며, 인접 화소에 반복적으로 접근하기 때문에 공유 메모리를 사용한다. 원본 프레임 *src*로부터 스크래치 화소를 계산하여 *spix* 행렬에 저장하는 GPU 커널 *detectScrPix*는 다음과 같다.

```
kernel detectScrPix(Mat src, Mat spix)
  int count = 0;
  shred_memory s_src[H][W] = src[H][W];
  int y = blockIdx;
  int x = threadIdx;
  for(int k=0; k<|Δ|; k++)
    if(s_src[y][x]-s_src[y][x+Δ]>theta_1)
      count++;
  spix[blockIdx][threadIdx]=count;
```

여기서, *Mat*은 행렬 $H \times W$의 행렬이다. *s_src*는 공유 메모리이며, *blockIdx*와 *threadIdx*는 스레드의 아이디를 결정하는 요소이다. $|\Delta|$는 Δ의 요소 수이고, *theta_1*은 스크래치 화소를 구분하는 임계값이며, *spix* 행렬에는 기준 화소와 X축으로 인접 화소 간의 차이가 급격한 경우의 수를 저장한다.

스크래치 세그먼트를 탐지하는 식 (2)는 하나의 스레드가 하나의 화소를 기준으로 하여 세로로 인접한 스크래치 화소의 수를 센다. $X \times Y$의 스레드를 구성하여 처리하며, 메모리 접근 시간을 줄이기 위해 공유 메모리를 사용한다. 스크래치 화소가 탐지된 행렬 *spix*로부터 스크래치 세그먼트를 계산하여 *sseg*에 저장하는 GPU 커널 *detectScrSeg*는 다음과 같다.

```
kernel detectScrSeg(Mat spix, Mat sseg)
  int count = 0;
  shred_memory s_src[H][W] = spix[H][W];
  for(int k=0; k<τ; k++)
    int y = blockIdx-τ/2+k;
    int x = threadIdx;
    if(s_src[y][x] == |Δ|)
      count++;
  if(count >= τ*0.7)
    for(int k=0; k<τ; k++)
      int y = blockIdx-τ/2+k;
      int x = threadIdx;
      sseg[blockIdx][threadIdx] = 1;
```

여기서, τ는 세그먼트의 길이이고, 기준 화소를 중심으로 하여 Y축으로 인접한 스크래치 화소의 수가 τ의 70% 이상일 경우 sseg 행렬의 해당 영역을 스크래치 세그먼트로 설정한다.

선형 스크래치를 탐지는 GPU에서 스크래치 세그먼트가 표시된 S_n의 *sseg* 행렬과 S_{n+1}의 *sseg* 행렬을 가상으로 연결하고, 하나의 스레드가 하나의 마스크 내에서 하나의 세로축에 대한 세그먼트의 길이를 조사한다. 그림 5와 같이 프레임의 행의 개수인 *(X × 마스크의 개수)* 만큼의 스레드를 구성하며, 공유 메모리를 사용하여 메모리 접근 시간을 줄인다. 스크래치 세그먼트가 탐지된 S_n의 *sseg* 행렬과 S_{n+1}의 *sseg* 행렬이 연결되어 구성된 가상 프레임 *svirtual*로부터 실제 선형 스크래치를 계산하여 *scr* 행렬에 나타내는 GPU 커널 *detectLinScratch*는 다음과 같다.

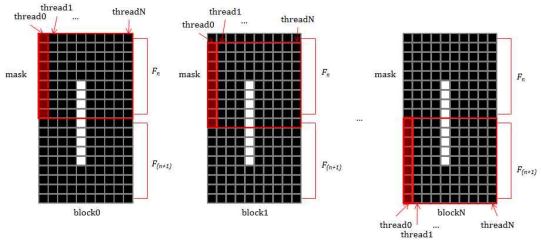

그림 5. 가상 프레임의 병렬 마스크
Fig. 5. Parallel mask of virtual frame

```
kernel detectLinScratch(Mat svirtual, Mat scr)
  int count = 0;
  shred_memory s_src[VH][VW] = svirtual[VH][VW];
  for(int k=0; k<H; k++)
    int y = blockIdx+k;
    int x = threadIdx;
    if(s_src[y][x] == 1)
      count++;
  scr[blockIdx][threadIdx]=count;
```

여기서, *VH*와 *VW*는 각각 가상 프레임의 가로와 세로 길이를 나타내고, *H*는 마스크의 세로 길이를 나타내며, *scr* 행렬

에는 기준 화소로부터 H까지의 영역 사이에 존재하는 스크래치 세그먼트 길이의 합이 저장된다. *scr* 행렬에서 세그먼트 길이의 합이 프레임 세로 길이의 일정 비율 이상일 때, 실제 선형 스크래치는 동일한 세로축 영역에 존재하는 스크래치 세그먼트들이며 해당 영역을 스크래치 세그먼트로 설정한다.

탐지된 스크래치를 복원하는 식 (3)은 얇은 스크래치의 경우 하나의 스레드가 하나의 스크래치 화소를 기준으로 하고, 두꺼운 스크래치의 경우 하나의 스레드가 두 개의 스크래치 화소를 기준으로 하여 인접 화소의 정보를 이용해 복원한다. $X \times Y$의 스레드를 구성하여 처리하며, 인접 화소에 대한 반복 접근 시간을 줄이기 위해 공유 메모리를 사용한다. 탐지된 스크래치 행렬 *scr*을 복원하여 *rst* 행렬에 저장하는 GPU 커널 *restorationScratch*는 다음과 같다.

```
kernel restorationScratch(Mat scr, Mat src, Mat rst)
  shared_memory s_scr[H][W] = scr[H][W];
  shared_memory s_src[H][W] = src[H][W];
  int y = blockIdx;
  int x = threadIdx;
  float avg;
  if(s_scr[y][x] == 1)
    avg = average(s_src[y][Δ]);
    rst[y][x] = minimum(absolute(s_src[y][Δ]-avg));
  else
    rst[y][x] = s_src[y][x];
```

여기서, *scr*은 스크래치 부분이 1로 설정된 행렬, *src*는 원본 프레임 행렬, *rst*는 스크래치가 복원된 프레임 행렬, *s_scr*과 *s_src*는 각각 *scr*과 *src*를 저장하는 공유 메모리이다. *s_scr*이 1이면 스크래치이므로 Δ 요소들의 화소 값 평균을 계산한 후, 각 요소들 중 평균과 가장 유사한 화소 값을 *rst* 행렬에 저장하며, 그렇지 않은 경우 *s_src*의 해당 화소 값을 저장한다. 복원된 프레임 행렬 *rst*는 host로 전송하여 저장한다.

IV. 실 험

제안된 알고리즘의 성능을 평가하기 위하여 국가기록원의 디지털화 영화 필름을 사용하였다. 실험 영상은 1960년대부터 1970년대 사이에 촬영된 것으로 아날로그 영상을 디지털로 변환한 영상이며, 객관적인 실험을 위해 컬러 영성과 흑백 영상을 혼합하여 10개의 영상을 선택하였다. 실험에 사용된 하드웨어의 사양은 Intel Core i7 3.4GHz, 8GB Ram, GTX580 GPU로 구성되었다. 제안된 알고리즘에서 사용한 임계값은 스크래치가 다수 나타나는 영상들을 사전 조사하여 식 (1)의 θ_l은 스크래치 화소와 정상 화소의 차이의 평균, 식 (2)의 τ는 스크래치 세그먼트 길이의 평균으로 설정했다.

순차처리와 병렬처리의 시간적인 차이를 비교하기 위해 각 영상에서 스크래치가 다수 발견된 연속한 1,000프레임씩 선정한 후 순차처리 연속 프레임과 병렬처리 연속 프레임 방법을 적용하여 전체 시간 T(Total), 준비 시간 P(Prepare), 연산 시간 O(Operation), 그리고 GPU를 사용하는 경우 데이터 전송 시간 TM(TransMission)과 GPU 연산 시간 GO(Graphic Operation)을 추가로 계산하였다. 시스템의 유휴상태 및 다른 프로세스로 인한 잡음 제거와 실험의 객관성을 위해 선정된 전체 1,000프레임과 절반인 500프레임에 대하여 각각 10회씩 적용한 후 전체 프레임 소요 시간과 절반 프레임 소요 시간의 차의 평균을 이용하였다. 처리시간은 하

표 1. 순차처리와 병렬처리의 수행 시간
Table 1. Computation time of sequential and parallel processing

V	순차처리 연속 프레임			병렬처리 연속 프레임				
	T	P	O	T	P	O	TM	GO
1	50.21	10.70	39.51	11.84	11.06	0.78	0.76	0.01
2	60.69	11.99	48.70	13.32	12.42	0.90	0.89	0.02
3	60.74	12.02	48.72	13.31	12.40	0.91	0.89	0.02
4	63.82	10.85	52.96	12.11	11.23	0.89	0.88	0.01
5	63.94	10.90	53.04	12.30	11.33	0.97	0.96	0.01
6	59.99	12.23	47.76	13.57	12.64	0.93	0.92	0.01
7	59.78	12.05	47.73	13.59	12.66	0.93	0.91	0.02
8	67.41	14.69	52.71	15.89	15.05	0.84	0.82	0.02
9	67.30	14.76	52.54	15.95	15.12	0.83	0.82	0.02
10	67.33	14.65	52.69	15.92	15.08	0.85	0.84	0.01
AVG	62.12	12.48	49.64	13.78	12.90	0.88	0.87	0.02

표 2. 탐지 및 복원 결과
Table 2. Detection and restoration results

V	스크래치 수	순차처리 단일 프레임		순차처리 연속 프레임		병렬처리 단일 프레임		병렬처리 연속 프레임	
		DR	M	DR	M	DR	M	DR	M
1	472	55.30	44.70	76.06	23.94	55.30	44.70	76.06	23.94
2	265	47.55	52.45	83.02	16.98	47.55	52.45	82.64	17.36
3	148	68.24	31.76	84.46	15.54	68.24	31.76	85.14	14.86
4	418	24.64	75.36	48.33	51.67	24.40	75.60	47.85	52.15
5	144	62.05	37.50	81.25	18.75	62.50	37.50	81.25	18.75
6	248	60.48	39.52	82.26	17.74	60.08	39.92	81.85	18.15
7	225	64.89	35.11	80.89	19.11	64.89	35.11	80.89	19.11
8	166	66.87	33.13	73.49	26.51	66.87	33.13	73.49	26.51
9	86	48.84	51.16	76.74	23.26	48.84	51.16	77.91	22.09
10	74	18.92	81.08	47.30	52.70	18.92	81.08	47.30	52.70
AVG	224.6	51.82	48.18	73.38	26.62	51.76	48.24	73.44	26.56

나의 프레임을 단위로 하며 표 1에 나타내었다. T를 보면 순차처리의 경우 프레임을 처리함에 있어서 평균 62.12ms 정도가 소요되며, 병렬처리는 평균 13.78ms 정도가 소요되었다. CPU가 관여하는 P를 제외한 O는 각각 평균 49.64ms와 0.88ms로 GPU를 이용하여 56.4배의 속도 향상을 보였다. GPU의 O의 대부분은 TM이기 때문에 공유 메모리를 사용하여 전송 시간을 줄여야 하는 이유를 보였다.

탐지 및 복원률에 대한 실험을 위해 각 영상에서 스크래치가 다수 발견된 연속된 100프레임씩 선정하여 총 1,000 프레임에 대하여 실험하였다. 먼저 선정된 프레임에 대하여 비전문가의 육안으로 스크래치를 탐지하여 기록한 후, 순차처리 단일 프레임, 순차처리 연속 프레임, 병렬처리 단일 프레임, 병렬처리 연속 프레임 방법을 각각 적용하여 탐지 및 복원율 DR(Detection and Restoration rate)과 미탐지율 M(Missed rate)을 표 2에 나타내었다. 순차처리 단일 프레임과 순차처리 연속 프레임 방법의 평균 DR은 각각 51.82%와 73.38%로 연속한 스크래치를 고려했을 때 20% 이상 탐지 및 복원율이 증가하였다. GPU를 이용한 병렬처리 방법에서는 병렬처리 단일 프레임과 병렬처리 연속 프레임 방법의 평균 DR이 각각 51.76과 73.44로 순차처리 방법과 유사한 탐지 및 복원율을 보였다.

IV. 결 론

제안하는 스크래치 탐지 알고리즘은 대량의 데이터 연산을 필요로 하기 때문에 CUDA를 이용한 병렬 설계를 통해 처리 속도의 향상을 시도하였다. 효율적인 병렬화를 위해 수많은 스레드로 작업을 분할하는 SIMT 구조의 모듈을 선택하여 병렬화 하고, GPU와 CPU간의 데이터 이동을 최소화 하였으며, 공유메모리의 사용을 통해 처리 시간을 단축시켰다. 실험에서 연속한 프레임의 스크래치를 고려하는 방법이 단일 프레임의 스크래치만을 고려하는 방법보다 20% 이상 우수한 성능을 보였으며, 병렬 설계를 통해 제안된 방법은 기존의 알고리즘과 탐지결과가 유사했으나 연산 시간은 50배 이상의 속도 향상을 보였다. 본 연구로부터 영상처리 알고리즘은 SIMT 기반 설계 가능성이 높아 병렬처리의 도입을 통한 높은 성능 향상이 기대된다. 그러나 대량의 데이터에 대한 GPU 전송에 걸리는 시간을 최소화 시키는 연구가 필요하다.

참고문헌

[1] D. Vitulano, V. Bruni, P. Ciarlini, "Line Scratch Detection on Digital Image: An Energy Based Model," WSCG'2002 10th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision'2002, pp. 447-484, Pilsen, Czech, Feb. 2002.

[2] Y. Ma, K. Xie, M. Peng, "A Parallel Gaussian Filtering Algorithm Based on Color Difference," IPTC, 2011 2nd International Symposium, pp. 51-54, Hubei, China, Oct. 2011.

[3] K. T. Kim, E. C. Ko, E. Y. Kim, "Digital Film Line Scratch Restoration based on Spatial Information," Korea Computer Congress, vol. 34, no. 1, pp. 454-459, June 2007.

[4] A. C. Kokaram, "Detection and Removal of Line Scratches in Degraded Motion Picture

Sequences," in Proc. Signal Processing VIII : Theories and Application, pp. 5-8, Trieste, Italy, Sept. 1996.
〔5〕 V. Bruni, D. Vitulano, "A Generalized Model for Scratch Detection," Image Processing, IEEE Transaction on, vol. 13, no. 1, pp. 44-50, Jan. 2004.
〔6〕 M. K. Gullu, O. Urhan, S. Erturk, "Scratch Detection via Temporal Coherency Analysis and Removal using Edge Priority Based Interpolation," in Proc. IEEE Intl. Symposium on Circuits and Systems, pp. 92-96, Island of Kos, Greece, May 2006.
〔7〕 B. M. You, K. T. Jung, S. K. Kim, D. S. Hwang, "Detection and Restoration of Vertical Non-linear Scratches in Digitized Film Sequence", in the 2012 Intl. Conf. on Image Processing, Computer Vision, and Pattern Recognition, Las Vegas, NV, July 2012.
〔8〕 NVIDIA, CUDA C Best Practices Guide, 〔online〕 Available: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.
〔9〕 David B. Kirk, Wenmei W. Hwu, Programming Massively Parallel Processors:A Hands-on Approach, Elsevier, pp. 99-103, 2010.
〔10〕 Jason Sanders, Edward Kandrot, CUDA By Example An Introduction to General-Purpose GPU Programming, Addison Wesley, pp. 79-81, 2010.
〔11〕 Marie Vans, Sagi Schein, Carl Staelin, Pavel Kisilev, Steven Simske, Ram Dagan, Shlomo Harush, "Automatic Visual Inspection and Defect Detection on Variable Data Prints," Journal of Electronic Imaging, vol. 20, no. 1, pp. Feb. 2011.
〔12〕 NVIDIA, NVIDIA Manufacturing Day 2013, 〔online〕Available: https://registration.gputechconf.com/form/session-listing.
〔13〕 Rob Farber, CUDA Applicatioin Design and Development, Elsevier, pp. 111-115, 2011.

저 자 소 개

이 준 구

2012: 단국대학교 컴퓨터과학과 공학사
현 재 : 단국대학교 전자계산학과 공학석사
관심분야: Image Processing, Parallel Processing, Machine Learning
Email : leejg01679@gmail.com

심 세 용

2013: 단국대학교 멀티미디어학과 공학사
현 재: 단국대학교 전자계산학과 석사과정
관심분야: Image Processing, Data Mining
Email : sheyong88@gmail.com

유 병 문

1987: 경북대학교 전자계산학과 공학사
1992: 충남대학교 전산과 이학석사
2000: Wayne State University 컴퓨터공학과 공학박사
현 재: ㈜엘앤와이비젼 대표이사
관심분야: Motion Analysis, 3D/2D image registration, (medical) image processing
Email : byoungmoon.you@gmail.com

황 두 성

1986: 충남대학교 이학사
1990: 충남대학교 이학 석사
2003: Wayne State University 박사
현 재: 단국대학교 컴퓨터과학과, 운동의과학과 부교수
관심분야: Machine Learning, Parallel Processing, Semantic Web
Email : dshwang@dankook.ac.kr