Journal of The Korea Society of Computer and Information
Vol. 20 No. 7, pp. 9-16, July 2015
www.ksci.re.kr
http://dx.doi.org/10.9708/jksci.2015.20.7.009

CUDA based parallel design of a shot change detection algorithm using frame segmentation and object movement

Seung-Hyun Kim*, Joon-Goo Lee**, Doo-Sung Hwang***

Abstract

This paper proposes the parallel design of a shot change detection algorithm using frame segmentation and moving blocks. In the proposed approach, the high parallel processing components, such as frame histogram calculation, block histogram calculation, Otsu threshold setting function, frame moving operation, and block histogram comparison, are designed in parallel for NVIDIA GPU. In order to minimize memory access delay time and guarantee fast computation, the output of a GPU kernel becomes the input data of another kernel in a pipeline way using the shared memory of GPU. In addition, the optimal sizes of CUDA processing blocks and threads are estimated through the prior experiments. In the experimental test of the proposed shot change detection algorithm, the detection rate of the GPU based parallel algorithm is the same as that of the CPU based algorithm, but the average of processing time speeds up about 6~8 times.

▸ Keyword : shot change detection, frame segmentation, Otsu's Threshold, CUDA

Ⅰ. Introduction

영상 기기의 활용이 증가하면서 고화질 영상의 기록물이 빠르게 증가하고 있다. 영상 데이터의 화소 수의 증가는 영상처리 시 높은 처리시간이 필요하다. 영상은 여러 장의 연속된 이미지의 집합이며, 멀티미디어 데이터 중 높은 저장 공간이 요구된다. 컴퓨팅 자원을 이용한 영상 데이터의 효과적인 관리는 영상 스트림의 분할(segmentation), 색인(index), 검색(search), 저장(store) 등의 관련 기술이 필수이다. 샷 전환 탐지(shot change detection)는 이를 위한 기반 기술이며 샷 전환 탐지에 대한 연구가 수행되었다. 샷은 하나의 카메라에서 촬영된 연속적인 프레임(frame)들의 집합이며, 샷과 샷 사이의 전환지점인 샷 경계(shot boundary) 또는 컷(cut)을 찾는 방법을 샷 전환 탐지이다.

범용(general)연산을 위해 설계된 CPU의 연산 시간은 처리 데이터의 크기 또는 연산량에 의존된다. GPU는 칩의 면적과 전력을 부동소수점 연산에 최적하기 위해 대량의 코어를 갖도록 설계되었으며, 많은 양의 연산을 분할 한 후, 각 코어에 분담시켜 계산하는 병렬처리 방법으로 계산 시간을 단축시키고 있다. 따라서 최근 GPU의 활용은 고 연산량이 요구되는 영상처리, 빅 데이터 분석, 생명 과학 등의 분야에서 사용되고 있다[1].

본 논문에서는 기 제안된 이동블록을 이용한 샷 전환 탐지 알고리즘[2]의 병렬설계를 제안한다. GPU를 활용한 병렬설계에는 NVIDIA의 병렬 컴퓨팅 아키텍처인 CUDA(Compute Unified Device Architecture)를 사용한다[3]. 제안하는 방법은 연산 시간을 단축하기 위해 샷 전환 탐지 알고리즘의 SIMT(single instruction, multiple threads) 구조의 특성을 갖는 Otsu 임계 값 설정[4,5], 프레임 이동, 프레임의 전역 히스토그램, 블록 히스토그램, 블록 비교연산을 병렬처리 설계를 제안한다.

본 논문의 2장에서는 기존에 연구된 샷 전환 탐지 알고리즘과 병렬처리를 활용하여 시간을 단축한 연구에 대해 토의한다. 3장에서는 제안하는 샷 전환 탐지 알고리즘의 병렬설계에 대해 설명한다. 4장에서는 샷 전환 탐지의 각 모듈별 시간측정을 통해 병렬설계의 성능을 측정하며,

• First Author: Seung-Hyun Kim, Corresponding Author: Doo-Sung Hwang
*Seung-Hyun Kim (pcpp13@naver.com), Dept. of Computer Science, Dankook University
**Joon-Goo Lee (leejg01679@etri.re.kr), Embedded System Research Section, ETRI(Electronics and Telecommunications Research Institute)
***Doo-Sung Hwang (dshwang@dankook.ac.kr), Dept. of Computer Science & Kinesiologic Medical Science, Dankook University
• Received: 2015. 06. 22, Revised: 2015. 06. 30, Accepted: 2015. 07. 07.

순차처리 결과와 비교한다. 마지막으로 5장에서는 제안 알고리즘의 장단점을 토의하고 향후 연구방향을 제시한다.

II. Related Works

Zhang 등[6]과 Xiaoquan Yi 등[7]은 연속된 두 프레임에서 대응되는 두 화소 값의 차이를 이용한 샷 전환 탐지 방법을 제안하였으며, Tak 등[8]은 RGB, HSV, YCrCb 히스토그램 정보의 차이를 이용한 샷 전환 탐지 방법을 제안하였다. 샷 전환 탐지 성능을 향상시키기 위해 프레임을 일정 크기의 블록으로 분할하여 블록의 차이를 이용한 방법들도 연구되었다. Go 등[9]은 블록 간 비교에 히스토그램 정보를 이용하였으며, Kang 등[10]은 블록 내 히스토그램을 기반으로 계산 된 Otsu 임계 값을 이용하였다.

블록화 기반 영역 정보를 이용한 샷 전환 탐지 방법은 프레임 전체 정보를 이용한 방법보다 샷 전환 탐지 성능은 향상되었지만, 연산량이 증가하여 높은 연산 시간이 필요하다. 영상처리는 프레임 또는 블록 내 화소 값을 기반으로 연산이 수행된다. 그러므로 영역 내 화소 정보 기반 동일한 연산 처리는 GPU의 SIMT 처리 방법에 적합하여, 관련 병렬설계 연구가 진행되었다.

Luo 등[11]과 Zhang 등[12]은 NVIDIA의 CUDA를 이용하여 엣지 탐지(edge detection) 알고리즘을 병렬로 설계하였다. CUDA 메모리 아키텍처 중 공유 메모리(shared memory)를 활용하여 Canny 엣지 탐지 알고리즘을 병렬처리하여 약 3.8배의 속도 향상을 보고하였다[11]. CUDA에서 공유 메모리는 온칩 프로세서에 있는 메모리로 SM(streaming multiprocessor)이 글로벌 메모리보다 빠른 데이터 접근을 보장한다. SM은 블록 단위로 프로그램을 처리하기 때문에 동일 블록 내의 스레드 사이에서만 공유메모리의 데이터에 대한 접근을 제공한다. CUDA의 텍스처 메모리(texture memory)를 활용하여 Sobel 엣지 탐지 알고리즘을 병렬설계하여, 영상 크기에 따라 최소 3.6배에서 최대 25.3배의 속도 향상 결과를 얻었다[12]. 텍스처 메모리는 캐시 읽기를 지원하며 읽기 전용 메모리이다. 그러나 CUDA SDK 2.2 버전 이후에는 텍스처 메모리를 지원하지 않는다[3].

샷 전환 탐지를 위한 히스토그램 연산의 병렬설계가 진행되었다[13,14]. 히스토그램은 프레임 내 화소들의 분포를 나타내는 것으로 병렬설계 과정에서 병목현상(bottleneck)이 발생할 가능성이 높다. 예를 들어 같은 값을 갖는 두 화소가 있다고 가정하면, 이를 처리하기 위해 두 스레드가 동일한 히스토그램의 빈 주소에 위치한 값을 증가시킬 때, 하나의 스레드가 먼저 빈을 증가 시키는 동안 다른 스레드는 지연 대기 시간이 발생하게 된다. Yang 등[13]은 히스토그램 평활화(equalization)를 병렬설계 하여 약 40배 이상의 속도 향상 결과를 얻었으며, 공유 메모리를 사용하여 병목 현상을 줄였다.

샷 전환 탐지에서도 알고리즘의 병렬설계를 통해 전체 시간을 단축시키려는 연구가 있다[15]. Lee 등[15]은 샷 전환 탐지 정확도는 유지하면서 프레임의 밝기 보상과 블록 간 차이 계산과정을 병렬설계 하였으며, 밝기 보상에서는 9.6배, 블록 간 차이 계산에서는 7.8배의 속도 향상 결과를 얻었다.

기 연구를 요약하면 GPU를 활용한 기 연구는 SIMT 식 병렬연산의 극대화와 속도 향상을 위해서는 데이터의 전송 시간을 최소화시키고 있다. 그러나 병렬처리를 이용한 연산시간보다는 CPU와 GPU간 데이터 전송 시간이 샷 전환 탐지 시간에 미치는 영향이 더 높다.

III. Proposed Method

GPU를 이용한 샷 전환 탐지 알고리즘의 병렬설계는 SIMT 구조를 갖는 연산을 CUDA 아키텍처를 이용하여 연산 속도를 향상시키는데 목적이 있다. 샷 전환 탐지의 병렬설계는 객체의 급격한 이동으로 인해 발생하는 오 탐지를 줄이기 위해 이동블록 간 변화를 탐지하며, 각 영상에 적합한 임계 값을 Otsu 임계 값 설정 방법을 이용하여 자동으로 추출한다. 히스토그램 연산 과정에서 발생하는 병목현상을 줄이기 위해 공유 메모리를 사용하였으며, 데이터의 전송 시간을 단축시키기 위해 CPU와 GPU사이의 데이터 이동을 최소화 시키는 병렬설계를 제안한다.

1. Shot change detection algorithm

샷 전환 탐지 알고리즘은 N개의 프레임으로 구성된 비디오 $V=\{F_1,F_2,F_3,\cdots,F_N\}$에 대해 샷 전환 탐지 결과 $D=\{d_1,d_2,d_3,\cdots,d_{N-1}\}$를 출력 한다. d_k는 프레임 F_k와 F_{k+1}사이의 샷 전환 여부를 판단한다. 그림 1은 제안하는 샷 전환 탐지 알고리즘의 처리단계이다. 각 영상에 적합한 임계 값 θ_1을 설정하기 위해 Otsu 임계 값 설정 방법을 이용한다. 전체 프레임에 대한 Otsu 임계 값을 계산한 후, 연속한 두 프레임의 Otsu 임계 값 차이가 3보다 큰 경우, 샷 전환 후보이다. 임계 값 θ_1은 영상 내 모든 샷 전환 후보군의 평균 Otsu 값으로 설정하며, 실험에서 블록의 샷 전환 여부를 결정하는 비교 대상으로 사용된다[4,5].

샷 전환 탐지를 위해 프레임 F_k를 $B_r \times B_c$개의 블록으로 분할하여 B_k^i에 저장한다. 프레임 내 모든 화소에 대해 블록 인덱스를 지정한 후, 각 블록의 히스토그램을 계산하여 H_k^i에 저장한다. 객체의 급격한 이동으로 인한 오 탐지를 줄이기 위해서 F_{k+1}의 대각선 방향 이동블록을 이용한다. 샷 전환 탐지를 F_k와 F_{k+1}의 블록 히스토그램을 비교하여, 5 개의히스토그램 차이 중 최솟값을 선택함으로써 객체의 이동으로 인해 발생하는 오 탐지를 줄일 수 있다.

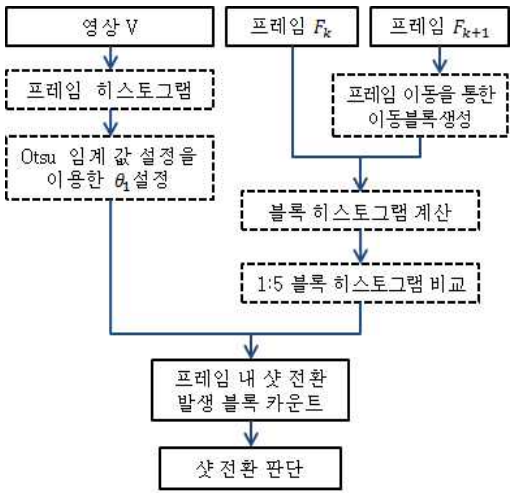

Fig. 1. The proposed shot change detection algorithm

선택된 최솟값을 θ_1과 비교하여, 블록의 샷 전환을 의미하는 b_i를 결정한다. 모든 블록에 대해 샷 전환 여부를 판단 한 후, 샷 전환이 발생한 블록의 수를 계산하여 임계 값 θ_2와 비교 한다. 샷 전환 발생 블록의 비율이 θ_2보다 크다면, d_k=1로 설정하고 F_k와 F_{k+1}사이에서 샷 전환이 발생한 것으로 탐지한다[2].

2. GPU based parallel design

제안하는 샷 전환 탐지 알고리즘의 병렬설계를 위해 NVIDIA의 병렬 컴퓨팅 아키텍처인 CUDA를 사용한다. CUDA는 NVIDIA의 GPU에서만 사용할 수 있으며, CPU와 GPU를 호스트(host)와 디바이스(device)로 구분한다. CUDA 기반 병렬설계의 성능을 최대화하기 위해서는 SIMT구조의 연산을 병렬설계 해야 하며, 호스트와 디바이스 간의 데이터 전송 시간을 최소화해야 한다. 여러 개의 스레드가 같은 주소에 접근할 때 발생하는 병목현상의 대기지연시간도 주의해야 한다[1,3].

본 논문에서는 샷 전환 탐지 알고리즘 중 프레임 히스토그램, Otsu 임계 값 설정 방법, 블록 히스토그램, 프레임 이동, 블록 간 히스토그램 비교 연산을 병렬로 설계하였으며, 각 연산의 GPU 커널을 의사코드(pseudo code)로 제시한다. 그림 1에서 병렬설계가 사용되는 설계단계는 점선 사각형으로 표시된다.

히스토그램은 각 화소 값의 누적 연산으로, 동일한 화소 값을 갖는 스레드들이 동일한 빈에 접근을 시도하여 병목현상을 발생시킬 가능성이 높다. 따라서 공유메모리를 사용하여 병목현상을 최소화하고 각 공유메모리의 연산 결과를 글로벌 메모리에 누적시키는 방법을 사용한다. 프레임의 전체 히스토그램을 계산하는 GPU 커널 histo_for_frame()은 그림 2와 같다.

```
kernel histo_for_frame(Mat frame, Hist rst ){
   Index i = blockIdx.x * blockDim.x + threadIdx.x;
   shared memory sub_hist [256 ] = {0 };
   atomicAdd(&sub_hist [frame [i ]], 1 );
   __syncthreads();
   if(threadIdx.x < 256 ){
     atomicAdd(&rst [threadIdx.x],
                              sub_hist [threadIdx.x]);
   }
}
```

Fig. 2. CUDA kernel **histo_for_frame()** calculating a frame histogram

그림 2에서 frame은 프레임, rst는 히스토그램을 의미한다. blockIdx.x, blcokDim.x, threadIdx.x는 CUDA에서 제공하는 내장 변수이다. 공유메모리는 CUDA 블록 내 스레드끼리 공유가 가능한 온 칩 메모리이며, 프레임의 행과 열의 수로 블록과 스레드를 설정한 경우 frame의 한 행에 대한 히스토그램이 공유메모리에 저장된다. histo_for_frame()에서는 sub_hist라는 공유메모리의 배열을 이용해 frame의 하나의 행에 대한 히스토그램을 계산한 후, 각 행의 히스토그램인 sub_hist들을 rst에 더해주는 방법으로 병목현상을 최소화 시킨다. 프레임을 일정 크기의 블록으로 분할 한 블록의 히스토그램을 계산하는 GPU 커널 histo_for_block()는 그림 3과 같다.

```
kernel histo_for_block(Mat frame, block_Hist rst
[B_r × B_c]){
   Index i = blockIdx.x * blockDim.x + threadIdx.x;
   Offset offset = blockDim.x;
   shared memory sub_hist [256] = {0};
   atomicAdd(&sub_hist [frame[i]], 1 );
   block Index blkIdx = (y/Y_b)*B_c + (x/X_b);
   __syncthreads();
   loop p from threadIdx.x to 256
      atomicAdd(&rst [blkIdx * 256 + threadIdx.x],
      sub_hist [threadIdx.x]);
      p += offset;
}
```

Fig. 3. CUDA kernel **histo_for_block()** calculating a block histogram

histo_for_block()은 histo_for_frame()과 다르게, 공유메모리에 프레임의 한 행에 대한 히스토그램이 아닌 블록의 한 행에 대한 히스토그램을 저장하기 때문에 블록 내 스레드의 수는 히스토그램 빈의 크기인 256 보다 작다. 따라서 누락되는 히스토그램 값이 없도록 공유 메모리의 모든 히스토그램 빈을 rst에 저장하기 위해 반복문을 사용한다.

Otsu 임계 값 설정 방법은 히스토그램을 두 그룹으로 나누는 임계 값을 구하는 방법이다[4,5]. Otsu 임계 값 설정은 데이

터 전송시간을 줄이기 위해 호스트의 메모리에 접근하지 않고, 그림 2를 통해 계산 된 디바이스에 존재하는 히스토그램 값을 사용한다. 커널의 블록과 스레드는 프레임 수와 히스토그램 빈의 수로 설정한다. Otsu 임계 값 설정 연산에 대한 GPU 커널 set_threshold_by_Otsu()는 그림 4와 같다.

그림 4에서 W는 가중치(weight), M은 평균(mean), V는 분산(variance), b는 배경(background), f는 전경(foreground), WCV는 클래스 내 분산(within class variance)이며, src는 GPU 메모리에 존재하는 영상의 모든 프레임에 대한 히스토그램, rst는 커널을 통해 계산된 WCV 값을 저장하는 변수, size는 프레임의 크기이다. Otsu 임계 값 설정의 마지막 단계인 최소의 WCV 값을 찾는 연산은 병렬설계에 적합하지 않기 때문에 호스트에서 처리한다.

```
kernel set_threshold_by_Otsu(Hist src, Otsu rst,
Size size){
  float Wb, Wf, Mb, Mf, Vb, Vf, WCV;
  loop t from 0 to 256
    if(t < threadIdx.x){
      Wb += src[t];
      Mb += (t * src[t]);
      Vb += (pow(t - Mb, 2 ) * src[t]);
    }
    else{
      Wb += src[t];
      Mb += (t * src[t]);
      Vb += (pow(t - Mb, 2 ) * src[t]);
    }
  Mb = Mb / Wb;      Mf = Mf / Wf;
  Wb = Wb / size;    Wf = Wf / size;
  Vb = Vb / size;    Vf = Vf / size;
  WCV = Wb * Vb + Wf * Vf;
  rst = WCV;
}
```

Fig. 4. CUDA kernel computing Otsu threshold **set_threshold_by_Otsu()**

GPU 기반 병렬설계에서 F_{k+1}의 이동블록 생성은 순차적인 프로세스를 사용하지 않고, 병렬설계를 위해 그림 5와 같이 F_{k+1}을 각 대각선 방향으로 λ만큼 이동시켜 총 4개의 이동 프레임을 생성시켜, 이동 프레임과 F_{k+1}를 $F_{k+1}(l)$, $(l=1,2,3,4,5)$에 저장한다. 인덱싱 순서는 우측상단부터 시계방향으로 진행하며, 마지막 $F_{k+1}(5)$에는 F_{k+1}를 저장한다. 이동 프레임을 생성한 후, $F_{k+1}(l)$을 $B_r \times B_c$개의 블록으로 분할하여 5장의 프레임으로부터 5개의 이동 블록을 생성하고 $B^i_{k+1}(l)$에 저장한다. 그림 5는 F_{k+1}을 우측 상단으로 λ만큼 이동시킨 이동 프레임의 예이다.

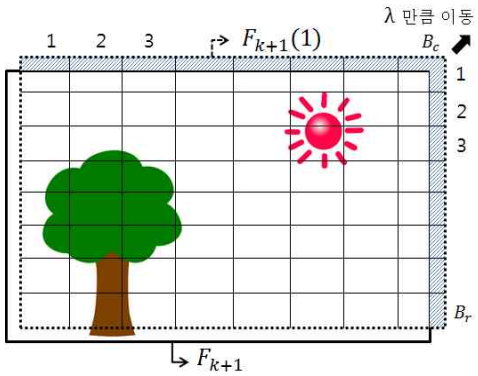

Fig. 5. The creation of $F_{k+1}(1)$ using the movement of F_{k+1}

그림 5에서 실선은 F_{k+1}, 점선은 $F_{k+1}(1)$이다. $F_{k+1}(1)$을 생성하기 위해 F_{k+1}을 우측상단으로 λ만큼 이동하는 과정에서 발생한 빗금 영역은 모두 0으로 채운다. 프레임 이동 연산은 하나의 스레드에 하나의 화소를 할당하여 새로운 위치로 이동시키기 때문에 병목현상이 발생하지 않아 병렬설계에 적합하다. 커널의 블록과 스레드는 프레임의 행과 열의 크기로 설정한다. 프레임을 이동시키는 GPU 커널인 move_for_frame()은 그림 6과 같다.

```
kernel move_for_frame(Mat src, Mat rst){
  Index i = blockIdx.x * blockDim.x + threadIdx.x;
  Num mvRow = blockRow * λ;
  Num mvCol = blockCol * λ;
  Index mvIdx = (blockIdx.x + mvRow )
        * blockDim.x +(threadIdx.x + mvCol );
  rst[mvIdx] = srt[i];
}
```

Fig. 6. CUDA kernel **move_for_frame()** for frame move operations

그림 6에서 move_for_frame()의 src는 이동 전 프레임, rst는 이동 후 프레임이다. mvRow와 mvCol은 λ에 의해 결정되며, 이동블록의 이동거리는 기존 샷 전환 탐지 연구[2]에서 실험을 통해 객체의 급격한 이동으로 인해 오 탐지 발생하는 경우의 평균 이동 비율인 블록 크기의 20%로 설정하였다. mvIdx는 화소의 이동 후 위치를 의미한다.

두 블록 간 히스토그램 차이를 계산하기 위해 디바이스에 있는 히스토그램 정보를 호스트로 전송하는 것은 데이터의 크기 측면에서 효율적이지 않다. 데이터의 크기를 줄이기 위해 그림 7과 같이 디바이스 상에서 블록 간 히스토그램의 차이를 계산한 후 각 빈의 값을 모두 합하여 호스트로 전송한다.

```
kernel compare_blocks(Hist hist,
                      Hist mv_hist[5], Num sum[5]){
  Index i = blockIdx.x * blockDim.x + threadIdx.x;
  shared memory comp[5][256];
  loop b from 0 to 4
    comp[b][threadIdx.x] = abs(hist[i]-mv_hist[b][i]);
    atomicAdd(&sum[b][blockIdx.x],
                                  comp[b][threadIdx.x]);
}
```

Fig. 7. CUDA kernel **compare_blocks()** comparing block histogram

블록 간 히스토그램 차이를 계산하는 GPU 커널 compare_blocks()은 그림 7에 제시하며, 커널의 블록과 스레드는 블록의 수와 히스토그램 빈의 수로 설정하여, 하나의 CUDA 블록에 한 개의 블록을 하나의 스레드에 히스토그램 빈 한 개를 할당한다. hist와 mv_hist는 히스토그램 변수이며, sum은 두 블록 간 히스토그램 차이의 합계를 저장하는배열이다. compare_blocks()을 사용할 경우 디바이스에서 호스트로 전송되는 데이터의 양은 약 1/307로 감소하였다.

Ⅳ. Experiment

본 논문의 실험은 제안하는 알고리즘의 병렬설계 성능 평가를 위해 GPU를 사용하였을 때와 사용하지 않았을 때의 샷 전환 탐지 정확도를 비교하고, 각 병렬설계 모듈 별 수행 시간 및 전체 시간을 측정한다.

실험에 사용된 PC의 사양은 Intel Core 2 Duo E7500 CPU 2.93GHz, GeForce GTX 480, 4GB RAM, 32비트 Windows 7 운영체제이며, OpenCV 2.4.9와 CUDA 6.5를 사용하였다. 실험 영상은 흑백 또는 컬러의 국가기록원 디지털화 영화필름[16], 드라마, 영화, 애니메이션이며, 영상의 정보는 표 1과 같다.

Table 1. The tested videos

타입	비디오	프레임 수	샷 전환 수
드라마 (컬러)	V1	5,000	55
	V2	5,000	67
영화 (컬러)	V3	5,000	51
	V4	5,000	79
기록원 (컬러)	V5	5,000	28
	V6	5,000	21
애니 (컬러)	V7	5,000	46
	V8	5,000	40
드라마 (흑백)	V9	5,000	67
	V10	5,000	61
영화 (흑백)	V11	5,000	59
	V12	5,000	79
기록원 (흑백)	V13	5,000	44
	V14	5,000	47

실험에서는 영상을 블록으로 분할하는 B_r과 B_c를 2^k, $k=1,2\cdots 5$까지 변화시켜, 샷 전환 탐지 비율이 높은 $k=3$으로 설정하였다. 임계 값 θ_1은 Otsu 임계 값 설정 방법을 통해 자동으로 추출하였으며, θ_2는 사전 실험을 통해 표본 영상에서 샷 전환이 발생한 경우 히스토그램 차이가 θ_1이상인 블록 개수의 평균인 49로 설정하였다. 이는 전체 블록 수 64의 77%에 해당하는 수치이다.

CUDA에서는 GPU 커널을 실행하기 위해서는 커널 개시와 동시에 블록과 스레드의 수를 설정해야 한다. 블록과 스레드의 적절한 크기를 얻기 위해 히스토그램 연산을 5,000번 반복하여 처리 시간을 측정하여 결과를 표 2에 제시한다.

Table 2. The processing time comparison of block versus thread sizes (unit : ms)

블록	스레드	5000회 처리시간	1회 처리시간
28,800	8	7,095	1.42
14,400	16	3,917	0.78
7,200	32	2,265	0.45
3,600	64	1,342	0.27
1,800	128	1,102	0.22
450	512	1,212	0.24
640	360	1,138	0.23

실험 결과 스레드가 128일 때 가장 좋은 효율을 보였으며, 64 이상부터는 큰 차이가 없었다. 실험에 사용된 GPU인 GTX 480은 CUDA의 페르미(Fermi)구조로 설계되어 2개의 워프가 스레드들을 스케줄링 하기 때문이다. 따라서 CUDA 스레드 인덱싱의 편리성을 위해 블록과 스레드를 프레임의 행과 열의 크기로 설정하였다. 병렬로 설계한 각 GPU 커널의 처리시간 측정 결과는 표 3과 그림 8에 제시한다.

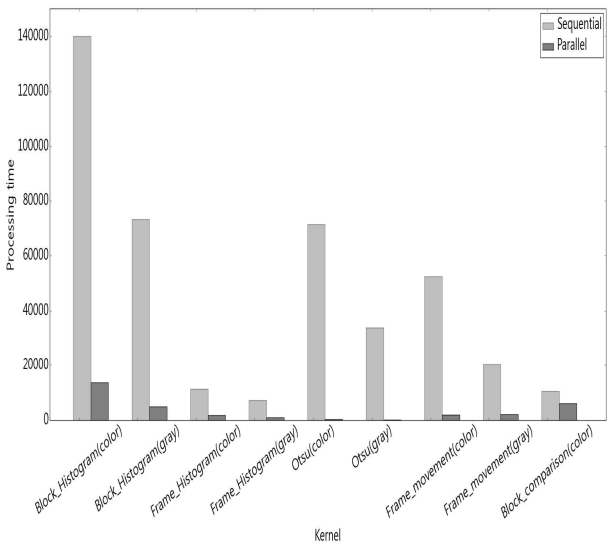

Fig. 8. The processing time comparison for CUDA kernel

Table 3. The processing time and speed-up ratio for CUDA kernels (unit : ms)

커널	순차처리	병렬처리	속도개선
블록 히스토그램	140,061	13,707	10.22
	73,279	4,955	14.79
프레임 히스토그램	11,353	1,800	6.31
	7,164	904	7.92
Otsu 임계 값	71,390	276	258.66
	33,872	92	368.17
프레임 이동	52,523	1,971	26.65
	20,278	2,194	9.24
블록 비교	10,459	6,140	1.70
	3,608	2,084	1.73

표 3에서 각 커널의 첫 번째 열은 컬러 영상, 두 번째 열은 흑백영상의 평균이다. 블록비교 커널의 경우, 속도 향상보다 호스트와 디바이스 간 전송되는 데이터의 양을 줄이는데 목적을 두었기 때문에 처리속도의 향상 비율이 다른 커널에 비해 낮다. 블록 비교를 제외한 GPU 커널의 경우, 약 6 ~ 350 배의 속도 향상 결과를 나타냈다. 각 GPU 커널의 호스트와 디바이스 간 데이터 전송 시간은 최소 3ms, 최대 2,123ms이었으며, F_k와 F_{k+1}의 샷 전환 탐지 알고리즘 한번에 소요되는 데이터 전송 시간은 평균 0.68ms이었다. 표 1의 실험 영상에 대한 샷 전환 탐지 비교 결과는 표 4와 같다.

Table 4. The shot change detection result of the proposed parallel design algorithm

타입	영상	Cut	순차 처리			병렬 처리		
			D	M	N	D	M	N
드라마 (컬러)	V1	55	55	0	0	55	0	0
	V2	67	67	0	0	67	0	0
영화 (컬러)	V3	51	51	0	0	51	0	0
	V4	79	79	0	0	79	0	0
기록원 (컬러)	V5	28	28	0	0	28	0	0
	V6	21	21	0	0	21	0	0
애니 (컬러)	V7	46	46	0	0	46	0	0
	V8	40	40	0	0	40	0	0
드라마 (흑백)	V9	67	67	0	0	67	0	0
	V10	61	61	0	0	61	0	0
영화 (흑백)	V11	59	59	0	2	59	0	2
	V12	79	79	0	0	79	0	0
기록원 (흑백)	V13	44	44	0	1	44	0	1
	V14	47	47	0	11	47	0	11

표 4의 D(detection)는 알고리즘이 탐지 한 샷 전환의 수, M(miss cut)은 실제로는 샷 전환이 발생했지만 알고리즘이 탐지하지 못한 샷 전환의 수, N(not cut)은 실제 샷 전환이 발생하지 않았지만 샷 전환이 발생했다고 오 탐지된 수이다. CPU만을 사용한 순차처리와 GPU를 사용한 병렬 처리의 샷 전환 탐지 결과 비교 시, 오차 없이 동일한 것을 확인할 수 있다.

히스토그램을 계산하는 커널 내부에 색상 변환 코드를 추가하여 직접 RGB를 GRAY로 변환하였다. RGB의 GRAY변환은 식 (1)을 통해 계산한다[17].

$$G = R \times 0.299 + G \times 0.587 + B \times 0.114 \quad (1)$$

식 (1)에서 G는 gray, R은 red, G는 green, B는 blue 값이다. GPU 커널 내부에 색상 변환 코드가 있기 때문에 OpenCv의 cvtColor() 연산을 병렬로 설계한 것이다. 데이터 전송시간을 포함한 전체 샷 전환 탐지 알고리즘 처리 시간은 표 5와 그림 9를 통해 제시한다.

Table 5. The processing time comparison of CPU versus GPU (unit : ms)

타입	영상	순차처리	병렬처리	속도개선
드라마 (컬러)	V1	282,927	37,407	7.56
	V2	276,334	37,491	7.37
영화 (컬러)	V3	280,335	37,904	7.40
	V4	276,893	37,634	7.36
기록원 (컬러)	V5	404,267	62,403	6.48
	V6	402,491	62,260	6.46
애니 (컬러)	V7	284,865	38,367	7.42
	V8	284,925	39,030	7.30
드라마 (흑백)	V9	171,370	27,180	6.31
	V10	172,642	27,302	6.32
영화 (흑백)	V11	170,136	27,588	6.17
	V12	170,743	27,644	6.18
기록원 (흑백)	V13	235,730	49,008	4.81
	V14	236,017	47,213	5.00
컬러영상 평균		311,630	44,062	7.17
흑백영상 평균		192,773	34,323	5.80
전체 평균		260,691	39,888	6.58

표 5의 흑백 영상의 속도 증가 비율은 RGB를 GRAY로 변환하는 과정에서 발생하는 시간으로 인해 컬러 영상에 비해 낮다. OpenCV의 cvtColor()을 사용 할 경우 평균 3.88배의 속도 향상이 있었지만, 색상 변환 코드를 GPU 커널 내부에 추가 할 경우, 평균 5.80배의 속도가 개선되었다. 컬러영상의 경우에는 색상 변환이 필요 없기 때문에 흑백 영상보다 높은 평균 7.17배의 속도향상이 있었다.

Ⅴ. Conclusion

본 논문에서는 Otsu 임계 값 설정 방법과 이동블록을 이용한 샷 전환 탐지 알고리즘에 대해 CUDA 병렬 아키텍처를 이용한 병렬설계 방법을 제안하였으며, 순차처리와의 비교를 수행하였다. 병렬설계 과정에서 발생하는 데이터 전송시간을 최소화하기 위해 한 커널의 연산 결과가 다른 커널의 입력데이터가 되도록 하였으며, Lee 등[15]과는 다르게 히스토그램 연산도 접근 속도가 빠른 공유 메모리를 이용한 병렬설계를 통해 처리 속도를 최대 14배 단축시켰다. 실험에서, GPU 기반 병렬 설계 알고리즘은 기존 순차처리보다 최대 7.56배의 속도 향상이 있었다. 본 논문의 연구 결과로부터 병렬설계가 SIMT 구조를 갖는 경우, 영상처리, 데이터 마이닝, 수치 해석 등 타 연구 분야에도 CUDA 병렬 아키텍처를 적용하여 속도를 개선 할 수 있을 것이다.

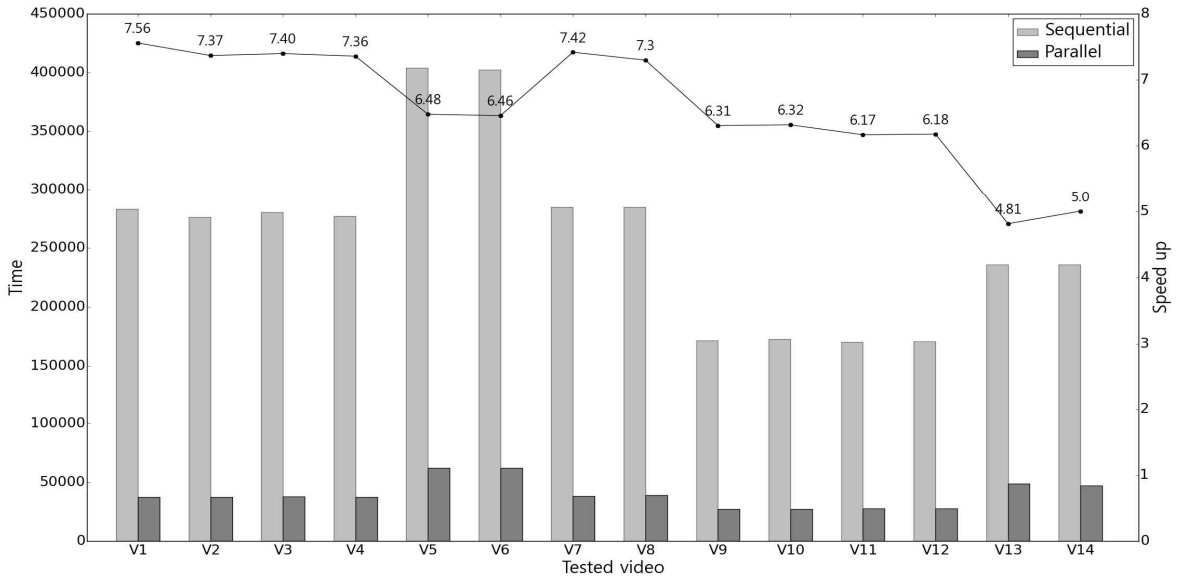

Fig. 9. The processing time comparison of CPU versus GPU

REFERENCES

[1] David B. kirk, Wen-mei W. Hwu, "Programming Massively Parallel Processors: A Hands-on Approach," Elsevier, pp. 73-105, 2010.

[2] S. H. Kim, D. S. Hwang, "A shot change detection algorithm based on frame segmentation and object movement," Journal of The Korea Society of Computer and Information, in press

[3] NVIDIA CUDA ZONE https://developer.nvidia.com/cuda-zone

[4] J. R. Parker, "Algorithms for Image Processing and computer vision 2nd Edition," wiley publishing, pp. 149-151, 2011.

[5] Puneet, Naresh Kumar Garg, "Binarization Techniques used for Grey Scale Images," International Journal of Computer Applications, Vol. 71, No. 1, pp. 8-11, June 2013.

[6] H. J. Zhang, A. Kankanhalli, S. W. Smoliar, "Automatic partitioning of full-motion video," Multimedia Systems, Vol. 1, No. 1, pp. 10-28, 1993.

[7] Xiaoquan Yi, Nam Ling, "Fast Pixel-Based Video Scene Change Detection," IEEE Int. Symposium on circuits and Systems, Vol. 4, pp. 3343-3346, May 2005.

[8] S. Y. Tak, S. Yoo, B. R. Lee, W. J. Lee, K. S. Ryu, T. G. Kim, H. C. Kang, "Scene change detection of various color space using difference of histogram," Proceedings of the Spring Conf. on The Korea Contents Association, pp. 466-468, May 2010.

[9] S. M. Go, H. G. Kim, M. S. Oh, "A Study on block histogram's comparison for cut detection," Journal of The Korean Institute of Maritime Information and Communication Sciences, Vol. 5, No. 7, pp.

1301-1307, Dec. 2001.
[10] S. J. Kang, S. I. Cho, S. J. Yoo, Y. H. Kim, "Scene Change Detection Using Multiple Histograms for Motion-Compensated Frame Rate Up-Conversion," Journal of Display Technology, Vol. 8, No. 3, pp. 121-126, March 2012.
[11] Yuancheng Luo, R. Duraiswami, "Canny edge detection on NVIDIA CUDA," Computer Vision and Pattern Recognition Workshops 2008. IEEE Computer Society Conference on, pp.1-8, June 2008.
[12] Nan Zhang, Yun-shan Chen, Wang Jian?Li, "Image parallel processing based on GPU," Advanced Computer Control (ICACC), 2010 2nd International Conference on, Vol. 3, pp. 367-370, March 2010.
[13] Z. Yang, Y. Zhu, Y. Pu, "Parallel Image Processing Based on CUDA,", 2008 International Conference on Computer Science and Software Engineering, Vol. 3, pp. 198-201, Dec. 2008.
[14] Thorsten Scheuermann, Justin Hensley, "Efficient histogram generation using scattering on GPUs," Proceedings of the 2007 symposium on Interactive 3D graphics and games(I3D), pp. 33-37, April 2007.
[15] J. G. Lee, S. H. Kim, B. M. Yoo, D. S. Hwang, "Parallel Design and Implementation of Shot Boundary Detection Algorithm," Journal of the Institute of Electronics Engineers of Korea, Vol. 51, No. 2, pp.76-84, Feb. 2014.
[16] National Archives of Korea, http://www.archives.go.kr/next/manager/preservation.do
[17] OpenCV document, http://docs.opencv.org/

Authors

Seung-Hyun Kim received B.S. in the department of Computer Science, Dankook University, Korea, in 2013. He expects to receive M.S. at the same university.

He is interested in parallel processing and image processing.

Joon-Goo Lee received B.S. and M.S. in Computer Science from Dankook University, Korea, in 2012, and 2014 respectively.

He is currently a researcher in Embedded System Research Section, ETRI(Electronics and Telecommunications Research Institute). He is interested in parallel processing, computer vision, embedded system and machine learning.

Doo-Sung Hwang received B.S. and M.S. from Chungnam University in 1986 and 1990 respectively, and Ph.D. in the department of Computer Science, Wayne State University, USA, in 2003.

He is currently a professor in Dept. of Computer Science, Dankook University. His interest is machine learning, parallel processing and image processing.