# GPGPU를 이용한 분리형 필터의 메모리 최적화[1)]

정윤혜, 김진우, 박용진, 한탁돈
연세대학교 컴퓨터과학과
e-mail: {yhjung, jwkim, jini99, hantack}@msl.yonsei.ac.kr

# Memory Optimization of Separable Filter Using GPGPU

Yunhye Jung, Jin-Woo Kim, Yong-Jin Park, Tack-Don Han
Dept. of Computer Science, Yonsei Univ.

요 약

분리형 필터는 메모리 접근 및 연산량을 $O(N^2)$에서 O(N)으로 최적화하는 기법이다. GPGPU를 이용한 필터기반의 영상처리에 분리형 필터를 적용하면 2차원 apron 영역이 2개의 1차원 apron 영역으로 이되기 때문에 메모리 최적화 측면에서 유리하다. 본 논문은 GPGPU를 이용한 분리형 필터에 슬라이딩 윈도우 형태의 지역 메모리 관리 기법을 적용하여 apron 발생을 최소화하는 구조를 제안하고 실험을 통해 저성능 메모리 환경에서 최대 17.9% 성능향상이 있었으며, 메모리 성능이 50% 낮아지는 경우에도 전체성능 감소도 최대 9%수준으로 안정적임을 확인하였다.

## 1. 서론

최근 스마트폰의 보급과 함께 모바일 기기에서의 고화소 영상 처리에 대한 수요가 증가하고 있다. 하지만 영상처리는 입력 데이터의 크기가 방대하며 연산량 또한 많기 때문에 저성능 기기에서 고화소의 영상을 실시간으로 처리하기 위해서는 가속이 필요하다. 영상처리를 가속하기 위한 다양한 방법이 연구[1][2][3]되고 있으며 최근에는 저성능 기기에서 고화소 영상처리의 성능을 높이기 위해 GPU를 이용한다.

GPU의 연산 성능은 이미 CPU의 성능을 능가하고 있으며 최근에는 GPU를 이용한 범용연산에 대한 환경의 지원으로 GPGPU의 응용 범위는 점차 증가하고 있다. 이러한 현상은 모바일 기기에도 점차 적용되고 있다.

GPU는 3D 그래픽스를 가속하기 위한 하드웨어로 높은 연산 성능을 가지고 있는 반면 단순한 렌더링 파이프라인을 기반으로 발전한 형태이기 때문에 흐름제어 측면에서 비효율적이다. 대부분의 영상처리는 제어보다는 연산이 많이 필요하며 출력되는 영상을 구성하는 각각의 화소가 모두 독립적이기 때문에 효율적인 병렬처리가 가능하다. 이러한 특징을 기반으로 영상처리 분야에 GPU를 적용하는 것은 효율적이다.

GPU는 일반적으로 메모리 성능보다 높은 연산 성능을 갖기 때문에 GPU를 이용한 영상처리에서 메모리 최적화 부분이 중요하게 된다.

대부분의 영상처리는 입력 영상에 제안된 알고리즘을 적용하여 필터 처리하는 회선처리를 따른다. 회선 처리는 영상 출력을 위해 주변에 위치한 화소 값을 고려하기 때문에 매번 계산을 할 때마다 중복적으로 메모리를 읽어야 하는 낭비가 발생한다. 따라서 메모리 접근을 감소하기 위한 방법으로 분리형 필터를 주로 사용한다[4]. 분리형 필터는 2차원 필터를 분리해서 사용하기 때문에 연산의 복잡도가 감소하고 메모리 접근 또한 줄어들게 된다.

본 논문에서 분리형 필터를 GPGPU에 효과적으로 적용하기 위해 GPU의 공유 메모리를 사용하여 중복된 메모리 접근을 최대한 제거하였으며 메모리 트랜잭션 감소를 위해 메모리 통합 접근(Memory Coalescing Access)을 적용하여 17.9%의 성능향상을 얻을 수 있었다.

본 논문의 구성은 다음과 같다. 2장에서는 분리형 필터에 대한 소개와 기존의 GPU를 이용해 분리형 필터에 대하여 설명한다. 3장에서는 GPGPU에 분리형 필터에서의 고려사항에 대해 설명한다. 그리고 4장에서는 제안한 방법을 실험을 통해 성능 검증 및 결과를 분석하고 4장에서 결론을 맺는다.

## 2. 관련연구

### 2.1 양방향 필터

양방향 필터(bilateral filter)[5]는 노이즈를 감소시키고

1) 이 논문은 2010년 정부 (교육과학기술부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임
(한국연구재단-2010-No.2010-0028259)

선명도를 증가시키는 비선형 필터이다. 양방향 필터는 두 개의 가우시안 필터(Gaussian filter), 도메인 필터(domain filter)와 레인지 필터(range filter)에 의해 동작하며 <수식 1>과 같이 표현할 수 있다.

$$I_P^b = \frac{1}{W_P^b}\sum_{q\in s} G_{\sigma_s}(\|p-q\|)\,G_{\sigma_r}(|I_p - I_q|)\,I_q$$

<수식 1> 양방향 필터

〈수식1〉에서 $W_P^b$는 정규화 인자를 나타낸다. $I_P^b$와 $I_q$는 각각 출력 영상과 입력 영상을 나타낸다. $G_{\sigma_s}(\|p-q\|)$는 도메인 필터를 나타내며 중앙 화소로부터 거리적으로 가까운 화소들에 대해 가중치를 부여한다. $G_{\sigma_r}(|I_p - I_q|)$는 레인지 필터로써 중앙 화소 값과 유사한 화소들에 대해 높은 가중치를 부여한다. $\sigma_s$와 $\sigma_r$은 각각 도메인 필터와 레인지 필터의 사이즈를 의미하며 값 조절을 통해 엣지를 보존하면서 노이즈를 감소시킬 수 있다.

## 2.2 분리형 필터

필터를 사용하는 영상처리는 주변의 화소 값을 함께 고려해 공간 영역을 계산하는 회선 처리 기법을 사용한다. <수식2>는 입력 영상에 2차원 회선처리 필터를 적용한 것이다.

$$r(i) = (s*k)(i,j) = \sum_n\sum_m s(i-n, j-m)k(n,m)$$

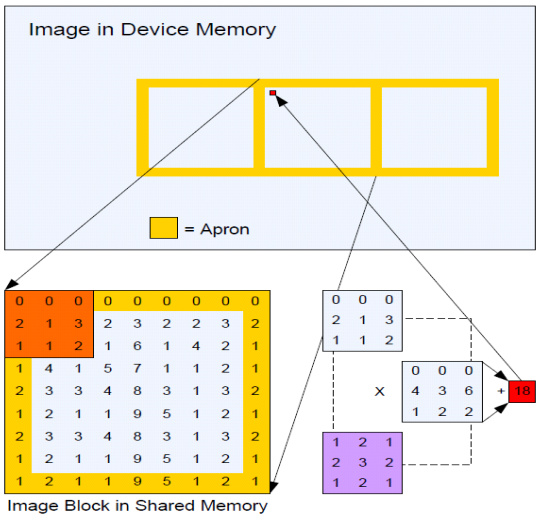
<수식 2> 2차원 회선처리 필터

<수식 2>에서 k는 입력 이미지, s는 입력 마스크, r은 회선 처리 된 이미지를 의미하며 2차원 회선처리 필터는 출력하고자 하는 픽셀 주위의 마스크 크기만큼 주변 화소에 가중치를 줌으로써 결과를 얻게 된다. 2차원 회선처리 필터는 마스크의 크기만큼 연산 및 메모리 접근이 증가하기 때문에 분리형 필터를 사용하여 이를 감소시킬 수 있다.

분리형 필터는 두 개의 1차원 필터, 즉 이미지의 행과 열로 구성되는 특별한 형태를 지닌다. 분리형 필터를 이미지에 적용하게 되면 연산 복잡도가 $O(N^2)$에서 $O(N)$으로 감소된다. 하지만 행과 열을 순차적으로 수행하기 때문에 중간 결과 값을 저장해야 하는 저장장소가 추가로 필요한 단점이 있다.

## 2.3 GPU에서의 분리형 필터

[4]에서 NVIDIA CUDA를 이용한 이미지 회선처리에 관한 연구는 분리형 필터를 적용한 것이다. 이미지를 블록 단위로 나누어 처리하고 공유메모리를 사용하여 쓰레드 간 데이터를 공유한다. 블록 구성은 (그림 1)처럼 처리할 이미지 블록의 커널 반경의 영역인 apron을 고려해 최적화해야 중복된 메모리 접근을 제거할 수 있다. 따라서 한 블록의 apron영역은 인접한 다른 block과 공유하게 된다. 하나의 쓰레드가 화소 하나를 처리하는 경우 apron 영역의 픽셀을 처리하는 쓰레드는 실제 필터 연산 시 수행을 하지 않는 상황이 발생하게 된다.




(그림 1) apron 영역의 고려

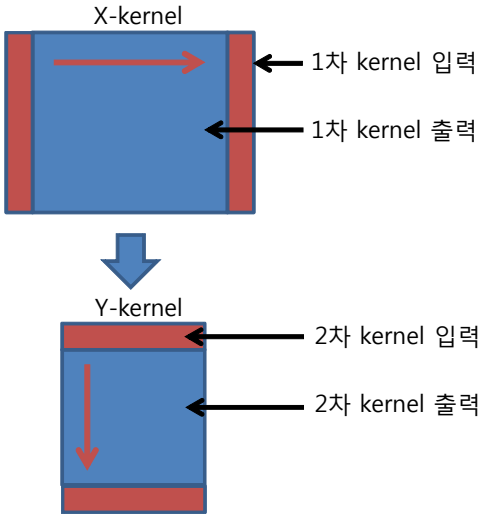
따라서 이러한 apron 영역을 줄이는 것이 중요하다. 또한 NVIDIA CUDA에서 메모리 최적화인 메모리 통합 접근의 적용을 위해서도 apron 영역을 half-warp(16) 단위로 구성해야 한다.

이러한 이점은 분리형 필터를 적용함으로써 더욱 커지게 된다. 예를 들어 행에 대한 필터를 쓰는 경우는 apron 영역이 양끝에만 존재하면 되기 때문에 메모리를 접근하는 횟수가 더욱 감소하기 때문이다. 이는 열 필터에서도 동일하게 적용된다. Apron을 가능한 줄이기 위해서 행 필터의 경우 가로축으로 길게 블록을 구성하고 열 필터는 세로축으로 길게 구성한다. 열 필터에서 블록의 형태가 가늘고 긴 것이 가장 좋으나 메모리 통합접근을 적용하기 위해 16의 배수로 구성해야 한다. 따라서 블록의 가로 길이는 16이 되어야 한다.

# 3. 제안하는 분리형 필터 최적화 구조

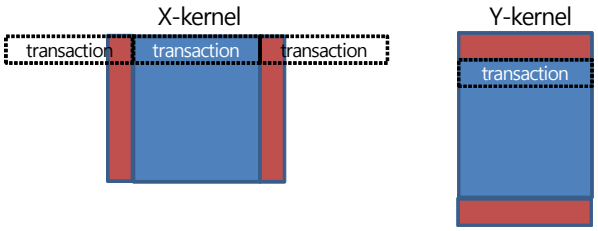
## 3.1 분리형 필터에서 고려사항

본 논문에서 적용하는 분리형 필터는 (그림 2)와 같이 X 커널과 Y커널로 이루어져 있다.




(그림 2) 분리형 필터의 구성

쓰레드 구성 시 추가적으로 요구되는 apron 영역의 접근을 최대한 줄이기 위해 X커널 경우 블록 구성 시 넓이를 최대화하고, Y커널은 높이를 최대화하여 구성한다. X커널과 Y커널을 분리해서 수행했을 경우 (그림 3)과 같이 Y커널의 성능이 좋은 것을 알 수 있다.

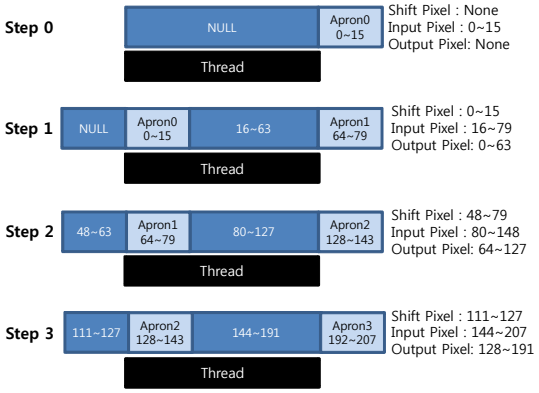



(그림 3) 분리형 필터의 apron 최적화

Y커널의 경우 가로의 길이가 트랜잭션을 완전히 활용 할 수 있도록 16의 배수로 알맞게 구성된 것을 볼 수 있다. 하지만 X커널은 결과 작업 그룹 이외에 처리 시 필요한 필터 크기의 apron 영역이 필요하다. 따라서 블록 당 양옆의 트랜잭션이 추가로 발생하게 되며 이때 불필요한 부분까지 메모리를 접근하게 된다. 이처럼 X커널의 경우 트랜잭션의 낭비가 생긴다. 따라서 본 논문에서는 X커널에서 발생하는 트랜잭션의 낭비를 줄이기 위한 최적화 방안을 제시한다.

### 3.2 공유 메모리를 이용한 최적화

본 논문은 분리형 필터를 적용한 영상처리에서 메모리 접근을 최대한으로 줄이는 것을 목적으로 한다. 이를 위해 (그림 4)와 같이 공유 메모리를 이용하여 다음 단계에서 사용할 메모리를 재활용함으로써 전역 메모리 접근을 줄인다.




(그림 4) 제안하는 지역 메모리 관리 구조

Apron 영역의 크기는 메모리 통합 접근이 허용 되도록 16으로 설정한다. 그림에서 위에 부분이 지역 메모리 상의 이미지 화소 값을 나타내고 아래에 위치한 쓰레드 영역이 실제 처리가 되고 있는 부분이 된다. 공유 메모리 부분을 살펴보면 이전 단계의 32개의 화소를 슬라이딩 윈도우와 같이 오른쪽으로 이동시켜 재사용하는 것을 알 수 있다. 따라서 64개의 화소를 처리하기 위해서는 96개의 화소 값이 요구되는데 매 단계마다 96개의 화소를 로드 하는 것이 아니라 64개의 화소를 새로 가져오고 32개의 화소를 이동하여 구성을 한다. 이와 같은 방법은 중복으로 발생하는 메모리 읽기를 줄임으로써 성능 향상을 이끌게 된다.

## 4. 성능 실험 및 최적화 분석

### 4.1 실험 환경 및 성능 측정 기준

4장에서 제안한 구조를 모바일 환경과 비슷한 수준으로 검증하기 위해 메모리 통합 접근 성능이 낮은 Compute Capability 1.0, 1.1을 지원하는 그래픽 카드를 이용하였으며 자세한 실험 환경은 <표 1>과 같다.

<표 1> GPGPU 실험 환경

| | |
|---|---|
| CPU | Intel i5 750 2.67 GHz |
| GPU | NVDIA 9400GT<br>- 32 shaders<br>- 4 CUDA multi-processor<br>- 275 MHz GPU clock<br>- 675 MHz Shader clock<br>- 200 MHz 64bit DDR2 Memory |

또한 다양한 GPU와 메모리 성능 환경 조건을 만들기 위해 RivaTuner[6]라는 GPU 제어 도구를 사용하여 메모리 클럭을 400MHz에서 200MHz까지 변경하여 실험하였다. GPU를 이용한 프로그램의 경우 3가지 측정 시간을 얻을 수 있다. 첫 번째는 CPU 동작 시간, 두 번째는 CPU와 GPU 사이의 메모리 전송 시간 그리고 세 번째는 GPU

동작 시간이다. 본 논문에서는 GPU 동작 시간만을 성능 비교에 고려하였다.

### 4.2 실험 내용 및 방법

4장에서 제안한 지역 메모리를 이용하여 분리형 필터기반 양방향 필터를 검증하기 위해 필터 전체 처리 시간과 최적화의 대상이 되는 X커널에 대한 성능 측정을 하였다. 일반적이 GPU최적화 기법[7]인 loop unroll 기법을 적용한 것도 함께 측정하였다.

### 4.3 양방향 필터 최적화 실험 및 결과

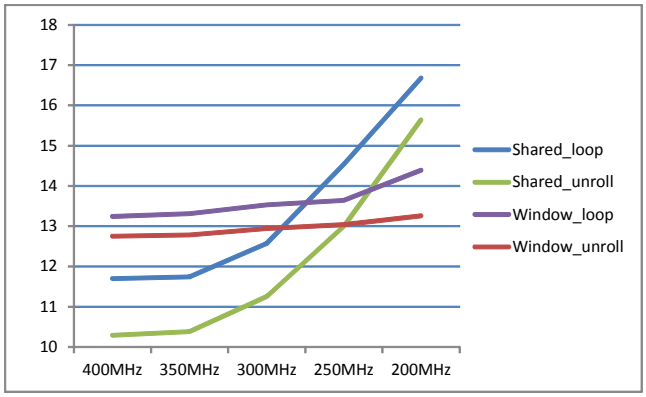
<표 2>은 분리형 필터가 적용된 양방향 필터를 지역 메모리에 사용한 결과를 나타낸다. <표 2>에서 기존의 방식은 Shared로 표기하였고 본 논문에서 제시하는 지역 메모리 기반의 슬라이딩 윈도우와 같은 메모리 관리를 수행하는 방식은 Window로 표기하였다.

<표 2> X커널 성능실험 결과(단위: fps)

| 메모리성능 / 실험 종류 | | 400 MHz | 350 MHz | 300 MHz | 250 MHz | 200 MHz | 성능 감소 |
|---|---|---|---|---|---|---|---|
| Sharedl/loop | | 85.5 | 85.2 | 79.6 | 68.8 | 60.0 | 43% |
| Shared/unroll | | 97.2 | 96.3 | 88.9 | 76.9 | 63.9 | 52% |
| Window/loop | | 75.5 | 75.1 | 73.9 | 73.3 | 69.5 | 9% |
| Window/unroll | | 78.4 | 78.2 | 77.3 | 76.7 | 75.4 | 4% |
| 성능 증가 | loop | -11.6% | -11.8% | -7.1% | **6.6%** | **15.9%** | |
| | unroll | -19.3% | -18.8% | -13.1% | -0.3% | **17.9%** | |

각각의 실험은 GPU 코어 클락을 275MHz로 고정하고 메모리 성능을 변화 시키며 진행하였다. apron 영역에 대한 최적화 없이 공유 메모리를 이용한 경우(Shared)도 메모리 통합 접근이 적용되었기 때문에 좋은 성능을 나타낸다. 메모리의 성능이 200MHz일 때 loop unroll이 적용되지 않은 경우 15.9% 성능 향상을 보였으며 loop unroll이 적용되지 않은 경우 17.9% 성능 향상을 보였다. 하지만 GPU 메모리 클럭을 최대치로 했을 때는 메모리보다 연산 성능이 부족하기 때문에 오히려 제안한 방법의 성능이 좋지 못한 것을 알 수 있다. 하지만 메모리 성능이 50% 감소하는 경우 기존 방법은 최대 52%의 성능 감소가 있는 반면 제안하는 구조는 최대 9%로 안정적인 성능을 보였다.

(그림 5)와 같이 본 논문에서 제시하는 방법은 메모리 성능이 좋거나 나쁜 경우 성능 차이가 거의 발생 하지 않는다. 따라서 저성능의 기기에서도 고성능일 때와 유사한 성능을 이끌어 내는 것을 알 수 있다.




(그림 5) X커널 성능실험 결과 그래프(단위: ms)

## 5. 결론

본 논문은 CUDA환경에서 분리형 필터를 구현할 때 메모리 접근을 최적화하는 방법을 제안하였다. 메모리의 성능을 제한하는 실험을 통하여 본 논문에서 제안하는 방식은 메모리 성능이 낮은 경우 기존의 병렬화 방식보다 효과적이며 전체성능이 메모리 성능에 영향을 적게 받는 것을 확인할 수 있었다. 제안하는 방식의 메모리 성능에 영향을 적게 받는 특성은 낮은 메모리 성능을 갖는 모바일 환경 등에 효과적으로 적용 될 수 있을 것이다.

### 참고문헌

[1] I. Haritaoglu, "Scene text extraction and translation for handheld devices", Proc. the IEEE Conferenceon Computer Vision and Pattern Recognition, vol.2, pp.408-413, 2001.

[2] C. Thillou, B. Gosselin, "Natural scene text understanding", Vision Systems, Segmentation and Pattern Recognition, Ch.16, pp.307-333, 2007.

[3] J. Zhang, A. Hanneman, A. Hanneman, J. Yang, A. Waibel, "A robust approach for recognition of text embedded in natural scenes", International Conference on Pattern Recognition, vol.3, pp.204-207, 2002.

[4] V Podlozhnyuk, "Image convloution with CUDA", NVIDIA Corporation white paper, 2007.

[5] Bombay, "Bilateral Filtering for Gray and Color Images", Sixth International Conference on Computer Vision, 1998

[6] RivaTuner, http://www.guru3d.com.

[7] Y Yang, P Xiang, J Kong, "A GPGPU compiler for memory optimization and parallelism management", ACM SIGPLAN Notices, 2010