CUDA를 이용한 웹캠 영상의 색상 형식 변환 최적화※

김진우, 정윤혜, 박진홍, 박용진, 한탁돈
연세대학교 컴퓨터과학과
{jwkim, yhjung, jhpark, jini, hantack}@msl.yonsei.ac.kr

Optimization of Color Format Conversion of WebCam Images Using the CUDA

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

요 약

웹캠은 영상 데이터의 전송시간을 줄이기 위해 메모리 정렬은 고려하지 않는다. 메모리 정렬이 되지 않은 영상 데이터는 GPU에서 처리하기 부적합하며 고속의 영상처리를 위해서는 GPU에서 메모리 최적화가 가능한 색상 형식으로 변환되어야 한다. 본 논문은 웹캠 영상의 색상 형식 변환을 NVIDIA CUDA를 이용하여 가속하는 최적화 기법을 제안한다. 메모리 접근과 쓰레드 구성에 대한 최적화를 진행하였고, 제안하는 구조의 성능 측정과 최적화 정도를 분석하기 위해 GPU 메모리와 연산의 성능을 제한하여 실험하였다. 그 결과 최적화 방법에 따라 최대 68% 이상 성능이 향상됐다.

ABSTRACT

Webcam doesn't perform memory-alignment in order to reduce the transmission time of image data. Memory-unaligned image data is unsuitable for the processing on GPU. Accordingly, we convert it to available color format for optimization in high speed image processing. In this paper, we propose a technique that accelerates webcam's color format conversion by using NVDIA CUDA. We propose an optimization which is about memory accesses and thread composition, also evaluate memory and computing performance for verifying a hypothesis which is the performance of the proposed architecture and optimizing degree on low-performance GPU. Following the optimization technique, we show performance improvements over maximum 68 percent.

Keywords : GPU, GPGPU, CUDA, Image Processing(영상처리), Color Format Conversion(색상 형식 변환)

접수일자 : 2010년 12월 27일 일차수정 : 2011년 01월 31일 심사완료 : 2011년 02월 07일
교신저자(Corresponding Author) : 한탁돈
※ 이 논문은 2010년 정부 (교육과학기술부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임.
(한국연구재단-2010-No.2010-0028259)

1. 서 론

디지털 영상 처리는 인간의 영상 인식을 돕거나 이차적인 응용을 위해 변환, 개선, 강조, 압축 등과 같이 영상을 가공하고 변형시키는 과정이다. 영상처리는 초기에 우주탐사 과정에서 얻은 영상의 품질을 개선하는 등의 특수한 목적으로 시작되었다. 이후 컴퓨터의 보급과 성능 향상으로 점차 통신, 기상, 의료, 산업 분야에서도 응용이 되었다.

근래에는 디지털 카메라와 카메라 폰이 대중화되고 스마트 폰이 등장해 디지털 영상 처리 기술 개발에 큰 영향을 주고 있다. 또한 모바일 기기 카메라의 화소수가 증가하는 추세와 맞물려 고화소의 영상을 실시간으로 처리하는 기술의 개발이 더욱 요구되고 있다.

고화소 영상의 실시간 처리는 메모리 성능과 연산처리 능력을 높이는 것이 핵심이다. 지금까지는 영상처리 전용 프로세서를 사용해 문제를 해결해왔지만 최근 GPU를 이용한 방식이 소개되고 있다[1,2]. GPU를 이용한 방식은 전용하드웨어를 이용하는 방식보다 성능이나 전력소모 면에서 비효율적이다. 하지만 프로그램 가능한 GPU는 다양한 기능을 쉽게 적용할 수 있다는 장점이 있다.

GPU는 3차원 그래픽스 처리를 위한 구조이기 때문에 범용 프로세서에 최적화된 알고리즘은 GPU에 적합한 알고리즘으로 변환이 필요하다. 특히 GPU를 이용한 병렬처리에서 메모리 접근은 병렬로 처리되기 때문에 각각의 쓰레드들의 메모리 접근 주소가 정렬되지 않으면 메모리 연산 효율이 감소하게 된다.

웹캠과 같이 메모리 정렬이 되지 않은 색상 형식의 영상을 GPU를 이용하여 처리하기 위해서는 메모리 정렬이 된 색상 형식으로 변환하여 처리하는 것이 효율적이다. 또한 GPU는 일반적으로 연산성능에 비해 제한된 메모리 성능을 갖고 있으며[3] 영상처리의 경우 픽셀단위의 연산처리이기 때문에 성능 개선을 위해서는 계층적인 메모리에 접근에 대한 최적화가 필요하다[4,5].

본 논문은 GPU를 이용한 영상의 색상 형식 변환을 메모리 접근과 쓰레드 구성 방법의 측면에서 최적화를 진행하였고 실험을 통해 최대 68%까지의 성능 향상을 확인했다. 또한 제안하는 방법의 최적화 정도를 분석하기 위해 GPU의 코어(core)와 메모리의 성능을 제한하는 실험을 진행하였다.

본 논문의 구성은 다음과 같다. 2장에서는 웹캠 기반의 영상처리에서 NVIDIA CUDA의 메모리 최적화 문제를 소개하며 3장에서는 메모리 최적화에 부적합한 색상 형식을 적합한 형태로 변환하는 구조를 제시한다. 4장에서는 제안한 구조들의 성능과 최적화 정도를 NVIDIA사의 GPU와 CUDA[6] 환경에서 측정하고 분석하며, 5장에서 본 논문의 결론을 맺는다.

2. CUDA 메모리 환경 및 최적화 방향

2.1 메모리 통합 접근

CUDA는 전역 메모리(global memory)와 공유메모리(shared memory)로 구성된 메모리 시스템을 제공하며 [표 1]과 같은 특징을 같는다. 일반적으로 NVIDIA CUDA는 고속의 공유메모리에 전역 메모리의 일부분을 저장하여 캐시 메모리의 형태로 이용한다.

[표 1] 공유메모리와 전역 메모리

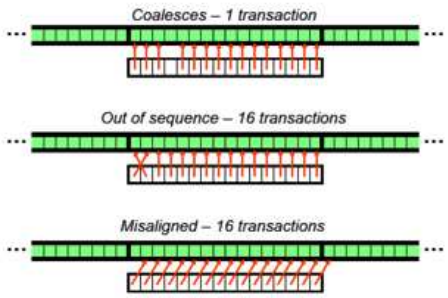
종류	용량	속도	접근
공유	16KB, 32KB	고속	블록
전역	256MB~2GB	저속	전역

전역 메모리의 데이터를 공유메모리에 미리 저장해 놓는 것은 다음과 같은 두 가지 특징이 있다.

첫 번째는 중복된 참조의 경우로 캐시 메모리와 비슷한 효과를 갖는 것이며 두 번째는 불연속적 메모리 접근 범위가 연속적인 형태로 전환될 수 있다는 것이다.

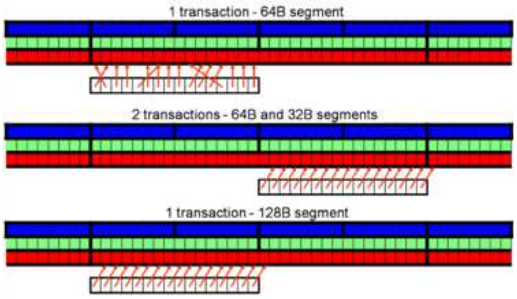
공유메모리에 복사하기 위한 전역 메모리 사용은 연속적인 접근이며 공유메모리 사용에서만 불연속적 접근이 발생한다. CUDA에서 각각의 쓰레드가 연속적인 공간의 메모리를 접근하는 경우 한 번에 최대 64바이트 또는 128바이트를 가져오기 때문에 메모리 성능을 크게 증가시킨다.

병렬처리 구조인 GPU는 병렬 수행되는 단위 쓰레드가 동시에 연속적인 메모리에 접근하는 경우 성능이 향상될 수 있다. 이를 메모리 통합 접근(memory coalescing access)이라고 하며 메모리 최적화에서 가장 중요한 부분이다. 메모리 통합 접근은 GPU의 종류에 따라 적용되는 범위 및 성능이 다르다.



[그림 1] 메모리 통합 접근1

[그림 1]은 NVIDIA GPU의 Compute Capability 1.0, 1.1에 적용되는 내용으로 첫 번째 경우는 16개의 쓰레드가 연속적으로 접근하여 한 번에 전송되었지만 두 번째와 세 번째 경우는 전체 또는 부분이 연속적이지 않아서 16번에 전송됨을 보여준다[5,7].



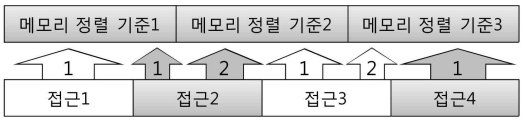
[그림 2] 메모리 통합 접근2

[그림 2]는 Compute Capability 1.2, 1.3에 적용되는 내용으로 쓰레드가 연속적으로 접근하지 않더라도 메모리 통합 접근이 적용됨을 알 수 있다[5,7]. 메모리 통합 접근은 GPU의 작업 스케쥴링 단위인 하프-와프(half-warp, 16개의 쓰레드)가 64바이트를 한 번에 가져오는 경우이며 이는 각각의 쓰레드가 4바이트 단위로 접근해야 함을 의미한다.

2.2 실시간 영상 데이터 형식의 문제점

CPU는 일반적으로 4바이트 단위로 메모리에 접근하기 때문에 대부분의 디지털 영상처리에서 메모리 정렬(memory alignment)을 위해 RGB 데이터에 1바이트를 패딩(padding)한 4바이트 크기의 픽셀을 사용한다. GPU도 이와 유사하지만 2.1절에서 설명한 메모리 통합 접근이 적용되기 때문에 16개의 쓰레드가 연속적인 64바이트를 접근해야 최적의 성능을 얻을 수 있다. 이는 각각의 쓰레드가 4바이트로 접근해야 하는 것을 의미한다.

모바일 기기의 카메라와 웹캠 등의 실시간 영상의 출력 영상이 4바이트 단위가 아닌 경우가 있다. 이는 외부 장치로의 데이터 전송 효율이 더욱 중요하기 때문에 4바이트로 정렬하여 전송하지 않고 3바이트 또는 그 이하의 단위로 전송한다. 이러한 경우 메모리 통합 접근은 불가능하기 때문에 4바이트 단위의 형식으로 변환할 필요가 있다.



[그림 3] 3바이트 픽셀의 4바이트 정렬

실험에서 사용한 웹캠은 마이크로소프트사의 HD-5000으로 RGB24형식을 지원한다. RGB24형식의 3바이트 픽셀을 공유메모리에 복사하는 것은 연속적인 접근이긴 하지만 쓰레드 그룹이 접근할 때 [그림 3]과 같이 메모리 정렬이 제대로 되지 않는다. [그림 3]은 블록이 전역 메모리에 접근할 때 가능한 4가지의 경우이다. 접근1은 시작 픽셀에 정렬이 되고 접근4는 마지막 픽셀에 정렬이 되지만 메모리 정렬 기준의 중간에 걸쳐있는 접근2, 3은 두 번의 접근이 필요하다.

3. 제안하는 색상 형식 변환 구조

2.2절에서 언급한 데이터의 형식이 다른 문제점 때문에 메모리 통합 접근을 적용한 최적화를 위해서는 입출력 데이터 형태와 쓰레드의 구성을 고려해야 한다. 입력 픽셀과 출력 픽셀의 크기가 다르므로 다양한 데이터 구성방법이 가능하며 쓰레드를 구성하는 과정에서도 입력과 출력의 다른 기준을 이용할 수 있다. 또한 성능이 제한된 환경에서 최적화를 분석하기 위해 Compute Capability 1.0, 1.1이 적용된 저성능 데스크탑 NVIDIA GPU를 사용하여 실험을 하였다.

3.1 입·출력 데이터 구성

웹캠의 RGB24형식은 3바이트이다. 메모리 통합 접근이 적용되기 위해서는 GPU의 메모리 접근 단위인 4바이트와 웹캠형식, 즉 3과 4의 최소공배수인 12바이트 단위로 접근해야 한다. 하지만 메모리 통합 접근이 가능한 64바이트는 12의 배수가 아니기 때문에 64와 12의 최소공배수인 192바이트 단위로 접근을 하는 경우 최적이 된다.

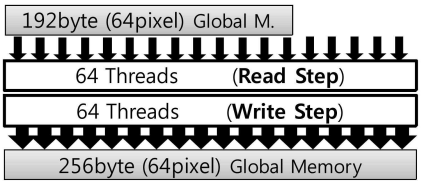
[그림 4] 192바이트 단위 픽셀 접근

[그림 4]는 NVIDA CUDA에서 제공하는 최적의 정렬기준과 제안하는 변환 메모리 접근 관계를 나타낸다. [그림 4]에서 정렬기준은 하프-와프(16개 쓰레드) 내의 각각의 쓰레드가 4바이트씩 접근하는 64바이트이다. 그리고 접근은 하프-와프가 3바이트 단위의 화소를 처리하기 위해 필요한 메모리의 양을 나타낸다. [그림 4]에서 쓰레드 블록이 192바이트(64픽셀) 단위로 처리하는 과정은 실제 연산에 필요한 192바이트를 중복 없이 한 번 읽었기 때문에 메모리 연산 효율은 100%로 가장 최적이 된다.

3.2 쓰레드 구성 최적화

입력 데이터를 3.1절의 최적 크기인 192바이트 단위로 구성하는 경우 4바이트 픽셀인 출력 데이터는 256바이트가 된다. 하지만 메모리 통합은 쓰레드 그룹의 입출력 크기가 같아야 하기 때문에 간단히 입력 또는 출력의 어느 한쪽에 맞춰 최적화를 하거나 입력과 출력을 전부 고려하여 쓰레드를 구성해야 한다. [그림 5]는 공유메모리를 사용하지 않는 경우의 알고리즘 및 쓰레드구성을 나타낸 그림이다. 4바이트 메모리 정렬과 하프-와프를 단순하게 적용하기 위해 출력 기준으로 쓰레드를 구성하였다.

```
Algorithm base<<<grid,block>>>(src, des, w, h)
{
        <화소 좌표 계산>
        <화소 입력 주소 계산>
        RGB = src[입력주소]
        <화소 출력 주소 계산>
        <RBG to YCbCr 변환>
        des[출력주소] = YCbCr
}
```

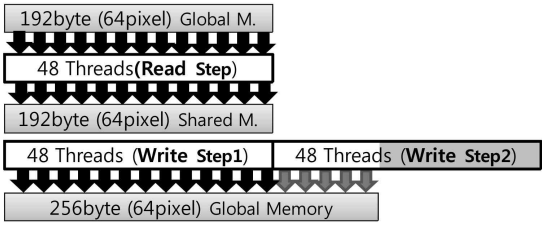


[그림 5] 메모리 최적화 되지 않은 구성

3.2.1 입력 기준 쓰레드 구성

입력만을 기준으로 고려하여 192바이트로 메모리를 정렬하는 경우 중복된 메모리 접근이 발생하지 않으므로 메모리 읽기 연산의 효율이 높다. 하지만 [그림 6]과 같이 48개의 쓰레드가 64개의 픽셀을 계산하고 저장하기 위해서는 쓰레드가 두 번 동작해야 하며 두 번째 동작에서 32개의 쓰레드는 유휴(idle) 상태가 되기 때문에 제어가 복잡해져서 비효율적이다. 48개의 쓰레드가 3번 동작하는 총 144번의 연산 동안 32개의 유휴 쓰레드가 발생하였기 때문에 쓰레드 효율은 78%이다.

```
Algorithm INPUT<<<grid,block>>>(src, des, w, h)
{
        __shared__ colorXY[blockX][blockY]
        <화소 좌표 계산>
        <화소 입력 주소 계산>
        <copy 전역 메모리 to 공유메모리>
        colorXY[threadX][threadY] = src[입력주소]
        쓰레드 동기화
        <화소 출력 주소 계산>
        <RBG to YCbCr 변환>
        des[출력주소] = YCbCr;
        IF(threadX<16)
                <RBG to YCbCr 변환>
                des[출력주소] = YCbCr
        ENDIF
}
```



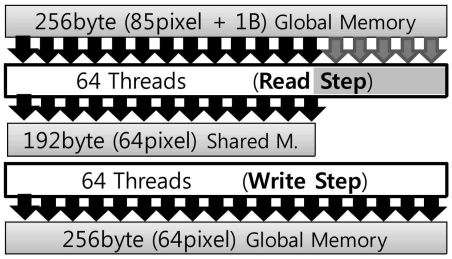
[그림 6] 입력 기준 쓰레드 구성

3.2.2 출력 기준 쓰레드 구성

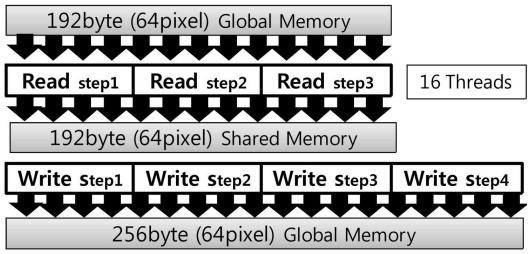
출력만을 기준으로 고려하여 256바이트로 메모리를 정렬하는 경우 85개의 픽셀을 읽어 64개의 픽셀만 저장하기 때문에 [그림 7]과 같이 중복된 메모리 읽기가 25% 발생하거나 16개의 쓰레드가 유휴상태가 된다.

64개의 쓰레드가 2번 동작하는 총 128개의 연산 동안 16개의 유휴 쓰레드가 발생하였기 때문에 쓰레드 효율은 87.5%이다. 또한 전체 쓰레드의 연산도 144번인 입력 기준보다 16번 적은 128번으로 11% 적다.

```
Algorithm OUTPUT<<<grid,block>>>(src, des, w, h)
{
        __shared__ colorXY[blockX*3][blockY]
        <화소 좌표 계산>
        <화소 입력 주소 계산>
        <copy 전역 메모리 to 공유메모리>
        colorXY[threadX][threadY] = src[입력주소]
        colorXY[threadX+16][threadY] = src[입력주소]
        colorXY[threadX+32][threadY] = src[입력주소]
        쓰레드 동기화
        <화소 출력 주소 계산>
        <RBG to YCbCr 변환> x 4
        des[출력주소] = YCbCr1
        des[출력주소+16] = YCbCr2
        des[출력주소+32] = YCbCr3
        des[출력주소+48] = YCbCr4
}
```



[그림 7] 출력 기준 쓰레드 구성



[그림 8] 입출력 기준 쓰레드 구성

3.2.3 입출력 기준 쓰레드 구성

입력과 출력을 모두 기준으로 고려하여 192바이트와 256바이트로 메모리를 정렬하는 경우 64픽셀을 한 번에 처리하는 쓰레드를 구성하면 192(3x64)바이트를 읽어 256(4x64)바이트를 쓰는 구조가 된다. 메모리 통합 접근이 가능하지만 입력과 출력의 크기가 다른 경우 [그림 8]과 같이 16개의 쓰레드로 작업그룹을 구성해야 한다.

이러한 구성에서 하나의 쓰레드는 3번 읽기 과정, 4개의 픽셀 처리 과정, 4번 저장하는 과정을 처리해야 하므로 쓰레드의 수행 시간은 길어진다. 하지만 전체 쓰레드의 수가 1/4로 감소하기 때문에 쓰레드 초기화 등의 작업을 줄일 수 있다. 또한 16개의 쓰레드가 7번 동작하는 총 112번의 연산이며 유휴(idle) 쓰레드가 발생하지 않기 때문에 100%의 쓰레드 효율을 갖는다.

```
Algorithm IN_OUT<<<grid,block>>>(src, des, w, h)
{
        __shared__ colorXY[blockX][blockY]
        <화소 좌표 계산>
        <화소 입력 주소 계산>
        <copy 전역 메모리 to 공유메모리>
        colorXY[threadX][threadY] = src[입력주소]
        쓰레드 동기화
        <화소 출력 주소 계산>
        <RBG to YCbCr 변환>
        des[출력주소] = YCbCr
}
```

[표 2]는 쓰레드 구성 기준에 따른 쓰레드 효율을 정리한 것으로 5장에서 실제 실험을 통해 성능을 비교하였다.

[표 2] 쓰레드 구성별 쓰레드 효율

기준	실제 연산수	실행 쓰레드	유휴 쓰레드	쓰레드 효율
입력	112	144	32	78%
출력		128	16	87.5%
입출력		112	0	100%

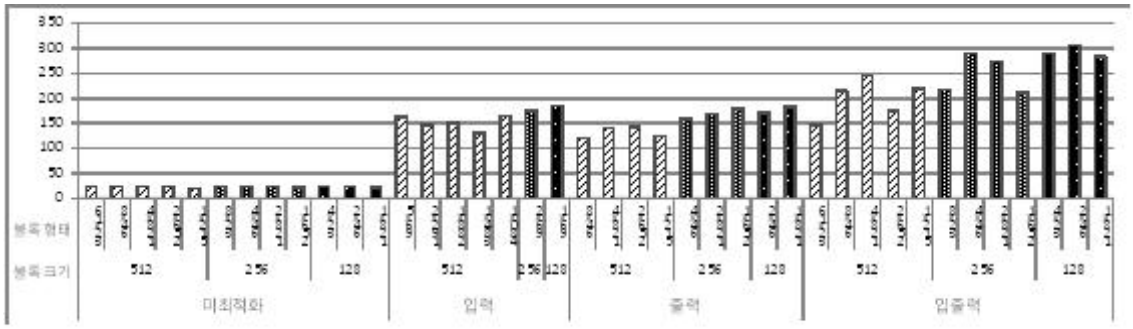
4. 성능 실험 및 최적화 분석

4.1 실험 환경 및 성능 측정 기준

제안한 구조를 성능을 검증하기 위해 메모리 통합 접근 성능이 낮은 Compute Capability 1.0, 1.1을 지원하는 그래픽 카드를 이용하였으며 자세한 실험 환경은 [표 3]과 같다.

[표 3] GPGPU 실험 환경

CPU	Intel i5 750 2.67 GHz
GPU	NVDIA 9400GT - 32 shaders - 4 CUDA multi-processor - 275 MHz GPU clock - 675 MHz Shader clock - 200 MHz 64bit DDR2 Memory
영 상	마이크로소프트사의 HD-5000 1280x720 웹캠 영상



[그림 9] 변환 기준에 따른 블록 형태별 성능 실험 결과

또한 다양한 GPU와 메모리 성능 환경 조건을 만들기 위해 RivaTuner[8]라는 GPU 제어 도구를 사용하여 GPU클럭을 550MHz에서 275MHz까지, 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 색상 변환 최적화 실험 내용 및 방법

4장에서 크기가 다른 색상 포맷의 변환을 위한 최적화 방법으로 입력과 출력의 기준에 따른 방법을 제안하였다. 제안한 구조를 검증하기 위해 24비트의 RGB 이미지를 32비트의 YCbCr로 변환하는 성능 실험을 수행하였다. 또한 GPU 성능 실험 시 제안하는 방법의 최상의 성능을 측정하기 위해 블록이 구성하는 그리드와 그리드를 구성하는 블록의 구성을 달리하여 실험하였다.

그리드 내의 블록 개수는 GPU의 멀티프로세서의 개수보다 많을 때 적어도 하나의 블록이 모든 멀티프로세서에서 실행될 수 있다. 또한 블록 당 쓰레드의 개수는 최적화된 연산 능력과 메모리 통합을 이용하기 위해 32의 배수가 되어야 한다. 하나의 멀티프로세서는 하드웨어 측면에서 768개가 동작하는 것이 가능하고, 블록 당 최대 포함할 수 있는 쓰레드의 개수는 512개이다. 예를 들어 하나의 블록이 256개의 쓰레드를 가지면 각각의 멀티프로세서에 3개의 블록이 할당된다. 이때 멀티프로세서 당 활성화된 와프(warp)의 비율을 의미하는 가동률(occupancy)은 100%가 된다.

블록 구성은 주로 실험 결과에 의한 어림법(heuristics)으로 이루어진다. 일반적으로 가동률이 50% 이상이면 성능 향상에 크게 영향을 미치지 않는다고 보며 128~256개의 쓰레드로 블록을 구성할 때 성능이 좋은 편이라고 본다[7].

따라서 본 논문은 다양한 블록 구성을 통해 성능 실험을 하여 제안하는 구조의 최상의 성능을 기준으로 평가를 하였다.

4.3 색상 변환 최적화 실험 결과

[그림 9]는 24비트 RGB를 32비트 YCbCr변환 과정을 각각의 최적화 방법을 이용하여 GPU에서 구현한 것과 CPU를 이용한 것의 실험 결과를 정리한 그래프이다. [그림 9]의 그래프에서 메모리 통합 접근 최적화를 적용하지 않고 전역 메모리를 이용한 구현은 '미최적화'로 구분하였다. 최적화되지 않은 경우는 공유메모리를 사용하지 않아 메모리 통합을 적용하지 않는다.

[그림 9]의 실험 결과는 각 변환 기준에 따라 블록의 형태별로 성능을 보여 준다. 입출력 기준으로 한 쓰레드의 구성의 경우 성능이 가장 좋다. 또한 블록의 크기에 따른 실험 결과를 보면 쓰레드

의 개수를 128개로 한정한 경우 평균적으로 성능이 좋은 것을 알 수 있다. [표 4]는 최적의 블록 구성을 정리한 표이며 [표 5]는 최적의 블록 구성으로 수행한 실험 결과를 정리한 표이다.

[표 4] 색상 변환 종류에 따른 블록 구성

종 류	블록X	블록Y	쓰레드 효율
입력 기준 구현	96	1	1
출력 기준 구현	128	1	1
입출력 기준 구현	64	2	1

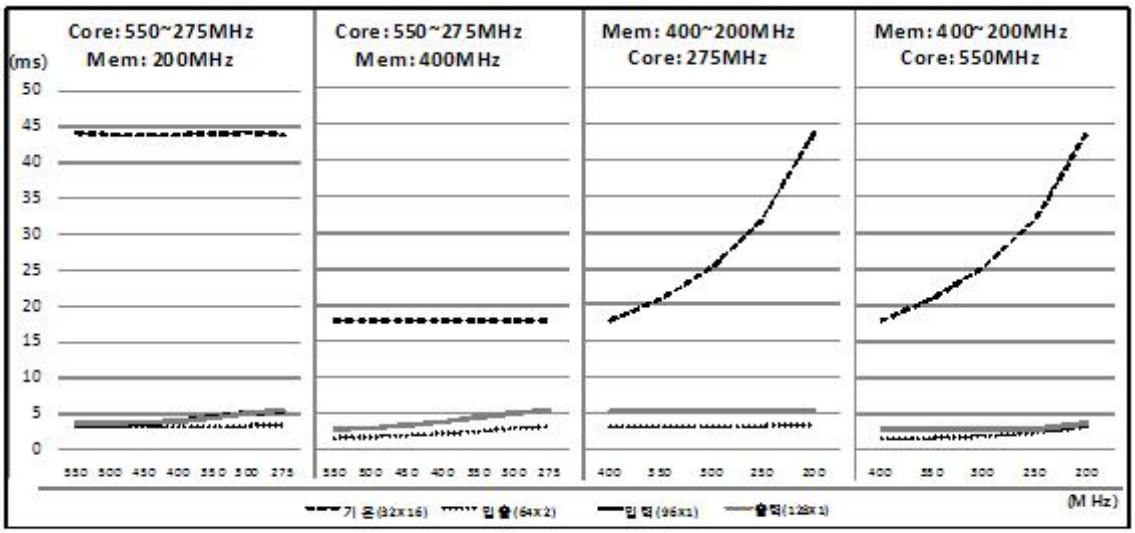
[표 5] 32비트 YCbCr 변환 성능 실험 결과

종 류	수행 시간	성능 (FPS)
입력 기준 구현	5.4243 ms	184
출력 기준 구현	5.4875 ms	182
입출력 기준 구현	3.2563 ms	306
GPU 미최적화	41.8945 ms	23
32비트CPU구현	25.6249 ms	39

GPU 구현에서 입력과 출력 모두 고려한 경우에 최상의 성능을 보였으며 출력 기준만 고려한 경우에는 최악의 성능을 보였다. 최상과 최악의 경우 최대 68%의 성능 차이를 보였다. 또한 32비트 CPU를 기준으로 최적화된 GPU 기법을 적용한 변환은 CPU보다 최소 4배에서 최대 8배 성능 향상을 보였다. 최적화되지 않은 GPU 기법을 적용한 색상 변환은 CPU보다 성능이 느리다. 이는 본 논문의 실험이 모바일에 적합한 환경을 갖추기 위해 GPU의 성능을 낮추었기 때문이다. 실험을 통해 본 논문에서 제안하는 색상 형식 변환 최적화는 상당한 성능 개선을 가져온다는 것을 알 수 있다.

4.4 최적화 정도 분석

[그림 10]은 제안하는 구조를 CPU와 메모리의 성능을 제한하여 실험한 결과이다. 첫 번째 그래프는 메모리 성능을 가장 낮은 200MHz로 고정하고 CPU 코어의 성능을 변화시킨 실험1의 결과이다. 두 번째 그래프는 메모리 성능을 가장 높은 400MHz로 고정하고 CPU 코어의 성능을 변화시킨 실험2의 결과이다. 세 번째 그래프는 GPU 코어의 성능을 가장 낮은 275MHz로 고정하고 메모리의 성능을 변화시킨 실험3의 결과이다. 네 번째 그래프는 GPU 코어의 성능을 가장 높은 550MHz로 고정하고 메모리의 성능을 변화시킨 실험4의 결과이다.



[그림 10] GPU/메모리 성능 제한에 따른 실험 결과 그래프

[표 6] GPU/메모리 성능 제한에 따른 전체 성능 감소

실험종류		실험1	실험2	실험3	실험4
성능 제한	메모리 성능 감소	200MHz	400MHz	400 → 200MHz(-0.5배)	
	GPU 코어 성능 감소	550 → 275MHz(-0.5배)		275MHz	550MHz
기준	미최적화(32x16)	0.00%	0.00%	59.64%	59.28%
	입출력기준(64x2)	48.80%	5.47%	49.09%	6.45%
	입력기준(96x1)	49.52%	38.90%	19.63%	0.54%
	출력기준(128x1)	49.38%	32.41%	24.94%	0.02%

[표 6]은 GPU 코어와 메모리의 성능을 각각 0.5배 감소시켰을 때 전체 성능변화율을 정리한 표이다. [표 6]의 실험에서 최대 0.5배 성능저하를 제한하였기 때문에 전체 성능의 감소는 암달의 법칙(Amdahl's law)에 의해 50%를 초과할 수 없다. 하지만 실험에 사용된 GPU는 실험에 필요한 연산 외에 영상출력 등의 운영체제의 기본적인 기능에도 사용된다. 따라서 제한된 GPU 코어와 메모리의 성능 제한은 0.5배 이상이 될 수 있으며 전체 성능의 감소도 50%를 초과할 수 있다.

미최적화(32x16)는 실험3과 실험4에서만 59% 이상의 성능저하가 있었기 때문에 메모리 성능에만 크게 영향을 받는 것을 알 수 있다. 이는 연산최적화를 통해 전체 성능향상을 기대할 수 없기 때문에 메모리 최적화가 필요함을 의미한다.

입출력기준(64x2)은 실험1과 실험2를 통해 GPU 코어 성능이 전체 성능에 주는 영향은 메모리 성능에 반비례하는 것을 알 수 있으며 실험3과 실험4를 통해 메모리 성능이 전체 성능에 주는 영향은 GPU 코어 성능에 반비례하는 것을 알 수 있다. 이는 연산 성능과 메모리 성능이 비슷한 수준으로 영향을 주며 균형적으로 최적화되었음을 의미한다.

입력기준(96x1)과 출력기준(128x1)은 입출력기준(64x2)과 비슷한 관계를 갖는다. 하지만 실험1과 실험2를 통해 메모리 성능이 높을 때 GPU 코어 성능에 적게 영향을 받으며 실험3과 실험4를 통해 메모리 성능의 영향이 입출력기준(64x2)보다 적음을 확인할 수 있다. 이는 쓰레드 수행과정이 입출력기준(64x2)보다 복잡한 것에 영향을 받는 것으로 판단된다.

GPU 코어와 메모리의 성능 제한 실험을 통해 색상 형식 변환은 GPU의 연산 성능 보다는 메모리 성능에 영향을 받는 것을 알 수 있다. 또한 이러한 영향은 메모리 최적화를 통해 제거할 수 있다.

5. 결 론

본 논문은 저장 형태가 다른 색상 형식을 GPU를 이용하여 변환할 때 그 과정을 최적화하기 위한 방법을 제안하였다. 입력과 출력 두 가지 기준으로 3가지 방법을 제안하였으며 성능과 최적화 정도를 분석하기 위해 CUDA를 이용하였다.

실험 결과에서 메모리 통합 접근이 적용되는 경우는 적용되지 않는 경우와 비교하여 최대 13배의 성능 향상이 검증됐다. 그리고 메모리 통합을 적용하는 과정에서 쓰레드 블록의 구성 형태에 따라서도 최대 68%의 성능 차이를 보였다. 이를 통해 GPU기반의 병렬처리의 최적화 과정에 메모리 통합 접근이 반드시 적용되어야 하며 쓰레드 내부의 알고리즘뿐만 아니라 쓰레드의 구성에 대한 고려도 반드시 포함해야 한다는 것을 알 수 있다.

GPU 연산과 메모리 성능의 제한 실험을 통해 GPU의 연산과 메모리의 최적화가 균형적으로 이루어 졌으며 실제 쓰레드 블럭의 구성이 GPU 연산 시간에 영향을 주는 것을 확인하였다. 또한 메모리의 성능을 제한한 경우에도 성능에 큰 변화가 없었기 때문에 메모리 성능이 낮은 모바일 환경에서도 제안하는 최적화 방법이 적용 가능하다고 볼 수 있다.

대부분의 GPGPU 관련 연구는 고성능 GPU를 이용하여 과다한 연산을 처리하는 것이 목적이다. 하지만 본 연구는 저성능 GPU 환경에서의 실시간 영상처리를 목적으로 하고 있다. 따라서 본 연구는 향후 모바일 환경에서도 CUDA와 OpenCL같은 고수준의 GPGPU 개발 환경이 제공되면 유용한 자료가 될 것으로 기대된다.

참고문헌

[1] N. Cornelis and L. Van Gool, "Fast scale invariant feature detection and matching on programmable graphics hardware", Proc. of Workshop on Visual Computer Vision on GPUs, June, 2008.
[2] Y. Luo and R. Duraiswami, "Canny edge detection on NVIDIA CUDA", Proc. of Workshop on Visual Computer Vision on GPUs, June, 2008.
[3] S Ryoo, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA", Proc. 13th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming, ACM Press, 2008.
[4] 김성수, 김동헌, 우상규, 임인성, “최적화된 CUDA 소프트웨어 제작을 위한 프로그래밍 기법 분석”, 정보과학회논문지 : 컴퓨팅의 실제 및 레터 제16권 제7호, 2010.7, pp 753-823(70pages).
[5] NVIDIA, "CUDA C Programming Guide for the CUDA Architecture Version 3.1", Aug 26, 2009.
[6] NVIDIA, http://www.nvidia.com/object/what_is_cuda_new.html.
[7] NVIDIA, "CUDA C Best Practices Guide Version 3.1", May 19, 2010.
[8] RivaTuner, http://www.guru3d.com.

김 진 우 (Jin-Woo Kim)

2006 상명대학교 소프트웨어전공 학사
2007-현재 연세대학교 컴퓨터과학과 석박통합과정

관심분야 : 컴퓨터 그래픽스 SW/HW, GPGPU, 렌더링 알고리즘

박 용 진 (Yong-Jin Park)

2003 연세대학교 컴퓨터과학과 학사
2005 연세대학교 컴퓨터과학과 석사
2005-현재 연세대학교 컴퓨터과학과 박사과정

관심분야 : 3차원 그래픽 가속기

정 윤 혜 (Yunhye Jung)

2009 성신여자대학교 멀티미디어전공 학사
2009-현재 연세대학교 컴퓨터과학과 석사과정

관심분야 : GPGPU, 영상처리 , 컴퓨터 그래픽스, 컴퓨터 비전, 렌더링 알고리즘

한 탁 돈 (Tack-Don Han)

1978 연세대학교 전자공학과 학사
1983 Wayne 주립대학교 석사
1987 매사추세츠 공대 박사
1989-현재 연세대학교 컴퓨터과학과 교수

관심분야 : 고성능 컴퓨터 구조, 미디어 시스템 구조, HCI (Human Computer Interface), 유비쿼터스 컴퓨팅

박 진 홍 (Jinhong Park)

2003 연세대학교 컴퓨터과학과 학사
2005 연세대학교 컴퓨터과학과 석사
2005-현재 연세대학교 컴퓨터과학과 박사과정

관심분야 : 2D/3D 컴퓨터 그래픽스 하드웨어, SoC 플랫폼