CUDA 기반의 공유 메모리를 이용한 Sobel 마스크 처리

조지훈*, 이상구**

요약

본 논문에서는 CUDA 기반에서의 공유 메모리를 이용하여 Sobel 마스크 처리기법를 구현한다. 기존의 공유 메모리를 사용한 Sobel 방법의 경우, replication을 위해 글로벌 메모리에 접근횟수가 늘어남에 따라 속도가 낮아진다는 단점이 있었다. 본 논문에서는 먼저 replication을 공유 메모리가 아닌 글로벌 메모리에서 수행한 후, 블록 복사를 통한 공유 메모리에 복사, sobel 연산을 수행하였다. 그 결과 기존의 공유 메모리만 사용한 알고리즘 보다 평균 30%의 성능향상을 확인 할 수 있었다.

Sobel Mask Operations Using Shared Memory in CUDA Environment

Ji-Hoon Jo*, Sang-Gu Lee**

ABSTRACT

In this paper, we implement Sobel mask operations using shared memory in CUDA environment. In the conventional method for mask operations using shared memory, there are some drawbacks of performance degradation as increasing the accesses on the global memory for image replications. Therefore, in this paper, we perform the replications operations on the global memory firtly and copy those blocks to the shared memory, and finally process Sobel mask oparations. The proposed algorithm minimizes the number of accessing to the global memory, so we get the higher speed-up factor about 30% as compared as the conventional algorithms.

Key Words : CUDA, Parallel processing, GPGPU, Sobel mask, shared memory

* 한남대학교 컴퓨터공학과(✉inver83@nate.com)
· 제1저자(First Author) : 조지훈 · 교신저자(Correspondent Author) : 이상구
· 접수일(2012년 9월 26일), 수정일(1차 : 2012년 11월 5일), 게재확정일(2012년 12월 18일)

Ⅰ. 서 론

2000년대 이후 단일 코어가 아닌 다중 코어를 장착한 CPU와 그래픽 처리를 위해 다수 코어를 장착한 그래픽 카드가 보편화 되었다. 그중에서 그래픽 카드는 단순히 그래픽 처리만이 아닌 다수 코어를 범용적으로 활용하는 GPGPU(general purpose GPU)가 등장하면서 병렬 처리 기법이 더욱 주목을 받고 있다.

CPU가 PC에서의 모든 명령어를 처리하는 핵심소자라 할 수 있다면, GPU(Graphic Processing Unit)는 컴퓨터 그래픽만을 전문으로 처리하는 연산장치를 뜻한다. GPU는 여러개의 코어를 바탕으로 병렬처리에 특화되었기 때문에 GPU의 연산 능력은 CPU와 비교하여 성능적인 측면에서 상당히 효율적이다. GPU는 다수의 코어를 제어하기 위해 SIMD(Single Instruction and Multiple Data) 아키텍쳐를 사용한다.

GPU가 다수의 코어 바탕의 고속 행렬계산이 가능하다는 점을 이용하여 기존의 3D 계산만이 아닌, 과학, 분자 시뮬레이션, 암호 분야에서도 활발히 활용되고 있다[1].

하지만, 기존의 GPU를 이용한 병렬처리 기법의 경우 그래픽스 파이프라인 상에서 이루어져야 하기 때문에 사용자들이 GPU를 이용한 파이프라인을 이해해야 효율적인 프로그래밍이 가능하다는 단점이 있었다. 이를 위해서 그래픽스 전문생산업체인 NVIDIA에서는 2007년 2월 GPU를 이용하여 범용적인 프로그램을 개발할수 있도록 프로그래밍언어, 컴파일러, 디버거, 라이브러리, 프로파일러를 한데 묶어 통합 개발 환경 'CUDA'를 발표하였다.

CUDA API는 C/C++ 기반으로 이루어져 있기 때문에 기존의 C/C++ 사용자들도 쉽게 병렬 처리 환경을 제공한다는 장점이 있다.

CUDA는 6가지 메모리 구조(글로벌 메모리, 공유 메모리, 상수 메모리, 텍스쳐 메모리, 로컬메모리, 레지스터)를 제공하고 있으며, 메모리의 종류에 따라 접근 속도 및 용량이 제한[1]되어 있기 때문에 병렬처리 구현시 알고리즘에 따른 메모리 설계가 요구된다. 더불어 CPU 기반의 프로그램을 GPU 기반의 프로그램으로 단순히 변환한다고 하여 성능상의 이점을 살릴 수는 없다. 메모리 구조에 따른 성능과 용량의 한계, 병렬 프로그램을 사용함에 필요한 동기화 기법, 지연감춤과 같은 고려해야 할 점이 존재하기 때문이다.

본 논문에서는 이미지 프로세싱 기법중 Sobel과 같은 마스크 기법을 CUDA 환경상에서 공유 메모리를 사용하여 구현한다. Sobel 연산의 경우 각각의 픽셀이 가로, 세로 필터를 사용하여 에지를 추출하기 때문에 병렬 처리 기법을 사용하는 데에 아주 좋은 방법이다.

본 논문의 구성은 다음과 같다. 2장에서는 CUDA의 메모리 구조 및 특징, 공유 메모리를 사용하기 위한 메모리 설계에 대해 알아본다. 3장에서는 실험 결과 및 분석에 대하여 알아보며, 4장에서는 결론을 다룬다.

Ⅱ. CUDA 메모리 구조 및 메모리 설계

2.1 CUDA 메모리 구조

CUDA는 속도와 크기가 다른 메모리 계층구조를 갖고 있으며, 각각은 메모리의 종류에 따라 속도 및 용량에서 차이가 난다. CUDA에서의 일반적인 처리 방법은 <그림 1> [2]에서 보는 것과 같다.

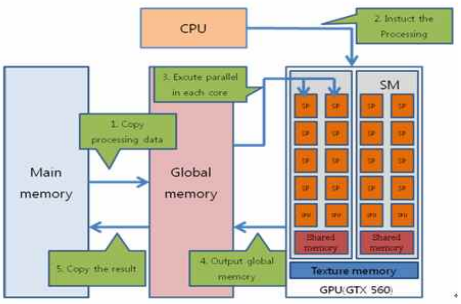

그림 1. CUDA의 명령처리
Fig. 1 Processing flow on CUDA

CUDA에서의 프로그래밍 구조는 먼저 병렬처리하고자 하는 데이터를 메인 메모리에서 GPU상의 전역 메모리(DRAM)으로 옮긴 후, 전역 메모리의 데이터를 다수의 쓰레드에서 처리한 후, 처리된 데이터를 GPU의 전역 메모리에서 메인 메모리로 가져오는 형태를 갖는다.

GPU의 내부에는 SP(Streaming Processor)라고 하는 작은 유닛이 있으며, SP는 GPU에서 사실상의 연산을 하는 코어이다. 이러한 코어 유닛은 Geforce GTX 560의 경우 448개의 코어를 갖고 있으며 하나의 SP는 CUDA 프로그램에서 4개의 쓰레드를 동작시키며, SP가 8개씩 모여 SM(Streaming MultiProcessor)의 집합으로 구성된다. CUDA의 경우 수백개의 쓰레드를 관리하기 위하여 SIMD(Single Instruction, Multiple Data) 아키텍쳐를 사용한다.

CUDA의 경우 관련서적 및 NVIDIA에서 제공하는 문서를 보면 메모리의 종류에 따라 성능 및 속도가 다르다[3]는 것을 알 수 있다. 일반적으로 많이 쓰이는 글로벌 메모리의 경우 용량적인 측면에서는 다른 메모리보다 우위를 점할 수 있지만, 속도 측면에서는 여타 다른 메모리보다 느리다는 단점이 있다. 일반적으로 CUDA 메모리 구조에서 가장 효과적으로 쓰일수 있는 메모리는 공유 메모리라고 볼 수 있다. 공유 메모리는 온 칩 프로세서에 있는 메모리로, 16KB의 용량을 L1 캐시와 동등한 속도로 사용할 수 있다. 공유 메모리를 읽고 쓰는 속도는 GPU 1 사이클 이내이다. CUDA에서는 SM당 16KB 크기의 공유 메모리를 Heap이나 Stack 메모리처럼 자유롭게 할당하고 사용할 수 있다.

2.2 공유 메모리를 이용한 Sobel 구현

일반적으로 CUDA에서 행렬곱, 내적, 백터의 합과 같은 문제를 해결할 경우, 쓰레드-블록 크기를 설계하는데 제약이 덜 하다고 볼수 있다. CUDA의 경우 하나의 블록당 512개의 쓰레드를 사용할수 있으며 블록들의 집합인 그리드의 경우 하나의 그리드당 65535개의 블록을 사용할수 있다. 가용할수 있는 쓰레드의 수가 512 × 65535 이기 때문에 일반적인 수학문제를 풀 경우 1차원으로 쓰레드를 배열하여 해결하는 것도 좋은 방법이다.

하지만, 이미지의 경우 쓰레드-블록 모델을 사용할 때에 위의 경우와 다르다고 볼 수 있는데, 그 이유는 첫 번째로 이미지는 2차원으로 이루어져 있으며, 두 번째로 Sobel, 캐니, 모폴로지와 같은 마스크 기법을 사용할 경우 이전 픽셀을 계속해서 참조해야 하는 경우가 발생하기 때문이다.

Sobel 연산[4]의 경우 보통 3 × 3크기의 2개의 마스크를 사용하여 가로, 세로를 미분하여 에지를 추출하는 연산을 뜻한다. 해당 픽셀을 기준으로 주변 픽셀의 값에 대하여 <그림 2> 의 필터를 차례로 값을 곱해 합을 구한 후, 그것의 절대 값들의 합을 결과로 리턴하는 형식을 띈다.

-1	0	1
-2	0	2
-1	0	1

-1	-2	-1
0	0	0
1	2	1

그림 2. Sobel 마스크 필터
Fig. 2 Sobel masks Horizontal, vertical filter

이와 같은 이미지 처리 기법은 각 픽셀별로 같은 연산을 하기 때문에 병렬성이 존재하며, CUDA를 이용할 수 있는 좋은 예 중 하나이다.

먼저, 각 픽셀에 대해 Sobel 연산을 위해 픽셀의 값을 전역 메모리(global memory)로 복사 한다. 공유 메모리(shared memory)를 이용하기 위해 전역메모리의 값들을 공유 메모리로 복사를 한다. 복사를 할때의 주의점은 공유 메모리는 16KB로 제한이 되어 있기 때문에 가용할 수 있는 쓰레드의 수에 제한이 있다는 것을 프로그래머가 숙지를 한후, 설계해야 한다.

본 논문에서는 512 × 512 픽셀의 Lena 이미지를 바탕으로 GPU를 이용하여 경계추출을 하기 위해 블록의 크기는 32 × 32, 각 블록별 쓰레드의 크기는 16 × 16로 하였다. 그 이유는 공유 메모리의 크기는 각 블록별로 16KB를 갖고 있으므로 8bit × 16 × 16 = 2KB로 제한하였다.

공유 메모리로 데이터를 복사하고 쓰레드 동기화 함수인 __syncthread() 명령어를 통해 각 블록별로 데이터가 공유 메모리에 옮겨지게 되면 공유 메모리에서 경계 검출 연산을 한다.

그림 3 와 같이 4 × 4 크기의 블록을 가정하자. 이 블록의 Sobel 연산 방법은 다음과 같다.

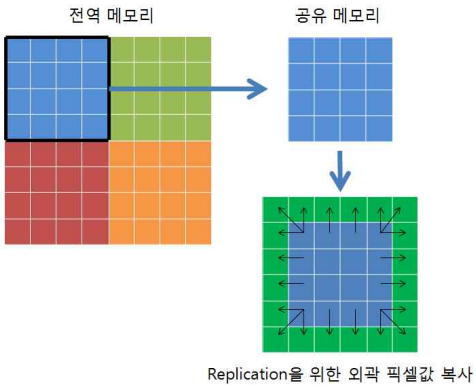

그림 3. 공유메모리부터 로의 복사

Fig. 3 Copying to Shared Memory

① Sobel 연산은 대상 픽셀 주위의 픽셀 값을 필요로 하기 때문에 그림 2와 같이 공유 메모리는 6 × 6 크기로 쓰레드 블록 크기보다 크게 할당되어야 한다.

② 이미지를 각 블록의 쓰레드의 크기만큼 나누어 공유 메모리에 값을 복사한다.

③ 만일 픽셀의 위치가 (x=1 || y =1) && (x= 4 && y=4) 일 경우, Replication 연산을 위해 블록의 외곽 픽셀 쓰레드의 값을 가져온다.

④ 각 블록별로 가로필터, 세로필터를 적용하여 Sobel연산을 한다.

⑤ __syncthread() 함수를 통해 각 블록별로 동기화 시켜준다.

⑥ 계산된 값들을 전역 메모리로 복사한후, 출력한다.

2.3 블록 복사를 이용한 Sobel구현

본 논문에서 제안하는 방법으로 블록별로 나누 기전 먼저 전역메모리에서 Replication을 사용하는 방법을 사용한다.

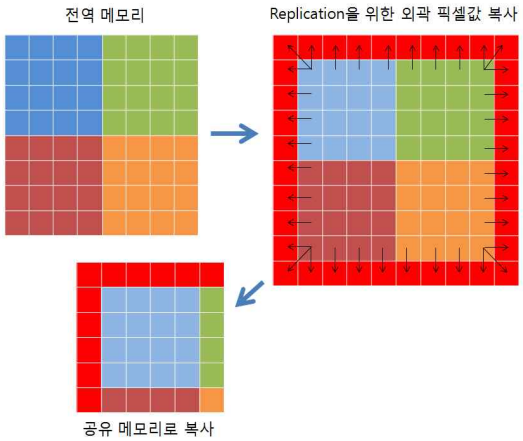

그림 4. 블록 복사를 사용한 Sobel 구현

Fig. 4 Sobel implemented using block copy

① 먼저 전역 메모리에서 Replication을 위해 x, y축을 2칸씩 늘려서 전역 메모리로 복사를 한다. (8 × 8의 블록이 10 × 10으로 늘려진다.)

② 6 × 6 크기의 블록으로 나누어 sobel 연산을 한다.

③ __syncthread() 명령을 통해 블록 별 동기화 한다.

④ Sobel 연산이 끝난후, 실질적으로 필요한것은 가장자리를 제외한 4 × 4 크기의 이미지 행렬이므로 인덱스 계산을 통해 공유 메모리를 가져온다.

```
예) for(i = x+1; i > x-1; x++)
        for(j > y+1; j > y-1; y++)
    sobel_operation(x,y);
```

end for
end for

⑤ 계산된 값들을 전역 메모리로 복사 후, 출력한다.

2.2의 방법이 Replication을 위해 각 블록별로 전역 메모리의 접근이 많았던 반면에 2.3의 방법의 경우 초반에 Replication을 위한 전역 메모리 접근 외에는 공유 메모리상에서 계산하기 때문에 속도 향상을 꾀할 수 있었다.

III. 실험결과

본 논문에서 제안하는 공유 메모리를 사용한 Sobel 구현을 실험하기 위한 환경은 표 1과 같다.

표 1. PC 개발환경
Table 1. PC developmet environment

	개발환경
H/W	CPU : Intel(R)Core(TM)2 2.0GHZ Memory : 2GB RAM Graphic card : NVIDIA GTX 560
OS	MS Windows XP Professinal SP3
실험도구	MS Visual Studio 2010, OpenCV NVIDIA CUDA Toolkit 4.2 NVIDIA Visual Profiler v4.2

표 2는 2.2에 설명하는 방법과 더불어 Sobel 연산을 각각 CPU상에서의 single 알고리즘을 적용하여 연산한 것, NVIDIA에서 예제로 제공하는 전역 메모리만을 사용, 공유 메모리만을 사용한 방법, 본 논문에서 제안하는 블록 복사를 사용한 방법에 대한 수행시간의 비교이다. 본 논문에서는 NVIDIA Geforce GTX 560 GPU에서 512 × 512 이미지에 대해 각각 100회씩 sobel 연산 이미지 처리에 대한 평균 수행 시간을 측정하였다.

표 2 에서도 볼 수 있듯이 제안된 최적화 기법으로 계산한 결과 일반적인 공유 메모리만 사용하는 방법보다 약 30% 성능향상을 확인할 수 있었다.

표 2. 각 알고리즘별 수행시간
Table 2. Comparison of each excution times

구현	실행 시간(sec)
CPU에서의 single 구현	2.5sec
전역 메모리만 사용할 경우	0.9153sec
공유 메모리만 사용할 경우	0.5531sec
제안하는 방법	0.3784sec

표 3 에서는 각 알고리즘별 분기발생수를 기록한 것이다.

표 3. 각 알고리즘별 분기발생수
Table 3. Number of occurrences of each quarter of the algorighm

구현	Branch	Divergence branch	instruction
공유 메모리만 사용할 경우	6581	102	67581
블록 복사를 사용한 sobel 방법	1987	19	54312

Branch 와 Divergence branch 및 instuction의 횟수를 측정하였다. Divergence branch[5]는 같은 warp 내부에서 일어나는 branch를 말하며 한 warp 에서 branch의 다음 연산을 기다리기 위해 쓰레드 들이 대기해야 하기 때문에 CUDA 성능에 좋지 않은 영향을 미친다. instruction 에서도 볼 수 있듯이 if문의 사용은 해당 쓰레드별 명령회수가 증가하므로 memory coalescing이 감소하여서 성능이 느리게 나오는 경우를 확인할 수 있었다. coalescing[6]은 GPU에서 동일

한 연산 수행을 위해 동시에 data를 가져오는 것이며 memory coalescing 의 횟수가 증가할수록 병렬처리에 유리한 알고리즘이다. 여기서 주목할 점은 모든 것을 공유 메모리를 사용하는 알고리즘이 전역 메모리 + 공유 메모리를 함께 사용하는 것보다 느리다는 점에 있다. 이는 공유 메모리만을 사용하는 방법의 경우 한 픽셀당, 9번의 읽기와 1번의 쓰기, 총 10번의 메모리 참조가 일어나고 각 쓰레드 동기화를 위해 __syncthread()를 사용하므로써 전체적인 속도 감소가 올 수 있다. __syncthread()는 각 쓰레드의 작업을 모두 동일하게 실행할 수 있도록 해주기 때문에 지연 감춤[7]의 효과를 떨어뜨린다.

일반적으로 공유 메모리의 경우 전역 메모리의 일부에서 데이터를 가져와서 처리 하는 방법을 많이 사용하지만, 공유 메모리에서 전역 메모리 참조를 위해서는 __syncthread() 함수를 사용하게 되고 이는 전체적인 성능 하향을 불러 올 수 있다.

IV. 결론

본 논문에서는 공유 메모리를 사용하여 sobel 연산을 구현하였다. 공유 메모리 사용시에는 공유 메모리의 속도가 저하를 가져올 수 있는(뱅크 충돌, 16KB의 제한된 용량) 여러 가지 요소들을 감안하여 설계하였다.

GPU상에서의 커널을 최적화하기 위해서는 해당 커널에서 메모리의 참조 횟수를 최소가 되도록 하였다.

본 논문에서 제안한 알고리즘은 이러한 요소들을 참조하여 전역 메모리 접근을 최소화하였으며, 이로 인해서 기존의 알고리즘에 비해 30%의 성능이 향상됨을 확인 할 수 있었다.

향후 연구로는 CUDA에서 제공하는 텍스쳐 메모리, 상수 메모리, 로컬 메모리 등을 활용 방법을 연구함으로써 최적화된 CUDA 프로그램을 연구할 필요가 있다.

참고문헌

[1] NVIDIA. NVIDIA CUDA Compute Unified Device Architecture: Programming Guide (Version 2.3), 2009.

[2] http://www.hbeongpgpu.com/whatiscuda.htm

[3] NVIDIA. *Optimizing CUDA*, 2009.

[4] Sobel, I., Feldman,G., A 3*3 Isotropic Gradient Operator for Image processing, Presented at a talk at the Stanford Artificial Project, 1968.

[5] Ali Bakhoda, George L. Analyzing CUDA WorkLodas Using a Detailed GPU Simulator, ISPASS-2009.

[6] Victor W Lee, Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU, ISCA'10, June 19-23, 2010, Saint-Malo, France.

[7] B. Parhami. Introduction to Parallel Processing: Algorithms and Architectures, Plenum Press, New York, pp.377-379, 1999.

감사의 글

본 논문은 2012년도 한남대학교 민군겸용보안센터의 지원에 의해 수행되었음.

저자소개

조지훈 (Ji Hoon Jo)

2011년 8월 : 한남대학교 컴퓨터공학과 졸업
2011년~현재 한남대학교 컴퓨터공학과 석사과정
※ 관심분야 : 객체추적, 영상처리, 영상 인식

이상구 (Sang Gu Lee)

1983년~현재 한남대학교 컴퓨터공학과 교수
※ 관심분야 : 영상처리, 임베디드 시스템, 패턴인식