최적화된 CUDA 소프트웨어 제작을 위한 프로그래밍 기법 분석

(Analysis of Programming Techniques for Creating Optimized CUDA Software)

김성수 [†] 김동헌 [†] 우상규 [†] 임인성 [††]
(Sungsoo Kim) (Dongheon Kim) (Sangkyu Woo) (Insung Ihm)

요 약 GPU(Graphics Processing Unit)는 범용 CPU와는 달리 다수코어 스트리밍 프로세서(many-core streaming processor) 형태로 특화되어 발전되어 왔으며, 최근 뛰어난 병렬 처리 연산 능력으로 인하여 점차 많은 영역에서 CPU의 역할을 대체하고 있다. 이러한 추세에 따라 최근 NVIDIA 사에서는 GPGPU(General Purpose GPU) 아키텍처인 CUDA(Compute Unified Device Architecture)를 발표하여 보다 유연한 GPU 프로그래밍 환경을 제공하고 있다. 일반적으로 CUDA API를 사용한 프로그래밍 작업 시 GPU의 계산구조에 관한 여러 가지 요소들에 대한 특성을 정확히 파악해야 효율적인 병렬 소프트웨어를 개발할 수 있다. 본 논문에서는 다양한 실험과 시행착오를 통하여 획득한 CUDA 프로그래밍에 관한 최적화 기법에 대하여 설명하고, 그러한 방법들이 프로그램 수행의 효율에 어떠한 영향을 미치는지 알아본다. 특히 특정 예제 문제에 대하여 효과적인 계층 구조 메모리의 접근과 코어 활성화 비율(occupancy), 지연 감춤(latency hiding) 등과 같이 성능에 영향을 미치는 몇 가지 규칙을 실험을 통해 분석해봄으로써, 향후 CUDA를 기반으로 하는 효과적인 병렬 프로그래밍에 유용하게 활용할 수 있는 구체적인 방안을 제시한다.

키워드 : GPU, 다수코어 프로세서, 병렬 프로그래밍, CUDA, 메모리 계층 구조, 지연 감춤, 점유율, 소벨 연산자

Abstract Unlike general-purpose CPUs, the GPUs have been specialized as many-core streaming processors, and are frequently replacing the CPUs in an increasing range of computations thanks to their outstanding parallel computing capacity. In order to respond to such trend, NVIDIA has recently issued a new parallel computing architecture called CUDA(Compute Unified Device Architecture), offering a flexible GPU programming environment for GPGPU(General Purpose GPU) computing. In general, when programmers use the CUDA API, they should clearly understand many aspects of GPU's computing architecture to produce efficient parallel software. In this article, we explain several optimization techniques for CUDA programming that we have verified through a lot of experiment and trial and error, and review how those techniques affect the performance of code execution. In particular, we use a specific problem as an example to analyze several elements that affect performances, such as effective accesses to hierarchical memory system, processor occupancy, and latency hiding. In conclusion, we present several directions that may be utilized effectively in CUDA-based parallel programming.

Key words : GPU, many-core processor, parallel programming, CUDA, memory hierarchy, latency hiding, occupancy, Sobel operator

---

· 이 논문은 2008년도 정부(교육기술과학부)의 재원으로 한국학술진흥재단(과제번호: KRF-2008-313-D00920)과 한국과학재단(과제번호: R01-2007-000-21057-0 (2009))의 지원을 받아 수행된 연구임

[†] 학생회원 : 서강대학교 컴퓨터공학과
kimss0418@naver.com
heuny85@sogang.ac.kr
coldnight.w@gmail.com

[††] 종신회원 : 서강대학교 컴퓨터공학과 교수
ihm@sogang.ac.kr

논문접수 : 2009년 12월 15일
심사완료 : 2010년 4월 27일

정보과학회논문지: 컴퓨팅의 실제 및 레터 제16권 제7호(2010.7)

1. 서 론

CPU의 다중코어(multi-core)와 GPU의 다수코어(many-core) 형태의 프로세서가 보편화됨에 따라 병렬처리 프로그래밍의 중요성은 점점 더 커지고 있다. 그동안 GPU는 그래픽스 계산 시 한 번에 수많은 기하 및 화소 데이터를 병렬적으로 처리하기 위해 코어의 수를 지속적으로 늘려왔으며, 연산 시 발생하는 메모리 접근 지연 시간을 숨기기 위해 쓰레드를 사용한 병렬처리로 계산 능력을 획기적으로 향상시켜 왔다. 이렇게 다수코어 기반 스트리밍 계산에 특화된 GPU의 연산 능력은 현재 CPU 대비 상당한 정도의 성능 우위를 점유하게 되었다[1].

이러한 특성으로 인하여 GPU는 방대한 데이터에 대해 동일한 계산 패턴을 요구하는 영역에서 사용이 늘어나고 있으며, 특히 과학계산 분야나 그래픽스 분야에서 CPU를 대신하고 있는 추세이다. 하지만 이러한 GPU 상에서의 프로그래밍은 특정 문제를 위하여 고안된 그래픽스 파이프라인 상에서 이루어져야 했기 때문에, 프로그래머는 효율적인 GPU 사용을 위해 추가로 그래픽스 파이프라인을 이해해야 하는 어려움이 따랐다. 이를 위하여 미국의 NVIDIA사에서는 GPGPU(General Purpose GPU) 아키텍처인 CUDA(Compute Unified Device Architecture)를 발표하여 그래픽스 파이프라인을 거치지 않고도 GPU를 활용할 수 있는 프로그래밍 환경을 제공하게 되었다.

CUDA API는 프로그래머에게 GPU 구조에 대한 특별한 지식이 없이도 C/C++언어와 유사한 문법을 제공함으로써, 프로그래머에게 친숙한 병렬 처리 개발 환경을 제공한다. 그러나 일반적으로 CPU 기반 프로그램에서 GPU 기반 프로그램으로의 단순한 변환만으로는 큰 성능 향상을 기대할 수 없다[2]. 알고리즘의 변환뿐만 아니라 계층적인 메모리 구조의 효율적인 접근이나 다수 코어들의 효율적인 사용 등을 위해서 반드시 알아야 할 점들이 있기 때문이다. 이런 측면에도 불구하고 일반 프로그래머들이 CUDA를 사용한 최적화 프로그래밍 기법들에 대해 쉽게 접근하기가 어려운 상황이며, 주로 많은 경험과 시행착오를 통하여 이해해야하는 상황이다.

본 논문에서는 본 논문의 저자들이 CUDA 프로그램 개발 과정 중 다양한 실험과 시행착오를 통하여 경험적으로 획득한 CUDA 프로그래밍 기법에 관한 정보를 제공함을 목적으로 한다. 이를 위하여 효과적인 계층 구조 메모리의 접근과 활성화 비율, 지연 감춤과 같은 문제에 대한 최적화 기법에 대하여 행렬과 벡터의 곱셈, 영상처리에 대한 실험 결과를 제시한다. 특히 CUDA 구조상 효율적인 메모리 접근이 용이하지 않은 특정 문제, 즉 소벨(Sobel) 연산 문제를 통해, 여러 가능한 최적화 기법의 적용이 프로그램의 효율에 어떠한 영향을 미칠 수 있는지 분석한다. 이러한 분석 결과는 향후 CUDA 프로그래머들이 시행착오를 최소화하면서 효율적인 소프트웨어를 개발하는데 도움이 될 것으로 예상한다.

본 논문의 구성은 다음과 같다. 2절에서는 개략적인 CUDA API의 프로그래밍 구조와 계층적인 메모리 접근 시 고려할 사항을 살펴보고, 3절에서는 코어의 점유율을 높이는 방법과 메모리 접근 시 지연 감춤 및 그 영향에 대해 기술한다. 4절과 5절에서는 GPU에서의 소벨 연산 적용을 통한 영상 처리 문제와 행렬과 벡터의 곱셈에 대하여 최적화 기법이 적용되는 과정 및 이후 실험 결과에 대해 설명한다. 마지막으로 6절에서는 결론에 대해 토의한다.

본 논문의 실험환경은 Windows Vista Service Pack 1 운영체제, 3.00GHz Intel Core 2 Duo CPU(E6850)와 2.00GB의 메모리, 그리고 NVIDIA GeForce GTX 280 GPU로 구성되어 있다[3].

2. CUDA 프로그래밍 구조 및 계층적 메모리 접근 최적화

NVIDIA사의 GPU는 속도와 크기가 다른 메모리 계층구조를 가지며 각각 다른 특징을 갖는다[4].

CUDA API를 사용한 일반적인 GPU 프로그래밍 구조는 처리하고자 하는 데이터를 CPU에서 GPU의 전역 메모리로 옮긴 후, 전역 메모리의 데이터를 GPU 상에서 처리하게 되며, 완료된 결과를 다시 GPU의 전역 메모리로부터 CPU로 가져오는 형태를 갖는다.

GPU는 몇 개의 프로세서 코어를 하나의 클러스터로 구성한 SM(Streaming Multiprocessor)의 집합으로 구성되어 있는데, NVIDIA GeForce GTX 280 GPU의 경우 30개의 SM이 존재하며, 각 SM은 8개의 SP(Scalar Processor) 코어로 구성되어 있다. 여러 프로그램에서 실행되는 수백 개의 쓰레드를 관리하기 위해 SM는 SIMT(Single-Instruction, Multiple-Thread) 아키텍처를 사용한다. SM 당 32개 쓰레드를 묶은 워프(warp)라는 단위로 SIMD(Single Instruction Multiple Data)형태의 연산을 수행하며 워프 단위로 명령어가 동시에 실행된다. 32개의 쓰레드는 8개 SP의 4단계의 연산자 파이프라인을 따라 4개의 8쓰레드 묶음으로 분할되어 흘러가게 된다.

2.1 메모리 접근 속도

NVIDIA사의 CUDA와 관련된 문서를 살펴보면 레지스터, 공유 메모리의 접근 속도가 가장 빠르며 상수 메모리, 텍스처 메모리, 그리고 전역 메모리와 지역 메모리의 순으로 느려진다[4]. 그러나 실제 프로그래밍 시 접근 속도가 빠른 메모리를 사용하였음에도 불구하고

더 느리거나 큰 성능향상이 나타나지 않는 경우도 발생한다[5].

2.2 메모리 접근 시 주안점

GPU는 워프 단위로 동시에 명령어를 처리하며, 메모리 참조가 발생할 경우 이는 다시 하프 워프(half-warp) 단위의 16개 쓰레드로 메모리를 동시에 참조한다. 각 메모리 계층 구조 접근 방법과 범위에 따라 성능에 큰 영향을 끼치게 되는데, 메모리 접근이 지연되는 예로 공유 메모리 접근 시 발생하는 뱅크 충돌(bank conflict)과 전역 메모리 접근 시의 메모리 접근 범위에 따른 트랜잭션 횟수 증가를 들 수 있다[1].

레지스터와 비슷한 접근 속도를 가지는 공유 메모리는 칩에 존재하는 메모리로써 4바이트 간격으로 16개의 뱅크로 구분되어 있으며 64바이트마다 이러한 패턴이 반복되어 나타난다. 쓰레드가 동시에 같은 뱅크 영역을 접근하려 할 경우 발생하는 문제가 뱅크 충돌로, 각 쓰레드는 병렬적으로 한 번에 메모리에 접근하지 못하고 같은 뱅크 영역의 충돌이 발생하지 않게 순차적으로 메모리 접근을 하게 된다[4](그림 1).

표 1은 3840×3840개의 데이터에 대해 공유 메모리 사용 시 각 하프 워프 당 16번의 뱅크 충돌이 발생하는 경우(BankConflict) 및 발생하지 않는 경우(NoBankConflict)에 대한 결과와 뱅크 충돌 부분을 제거한 결과를 NVIDIA 프로파일러(NVIDA CUDA Visual Profiler Version 2.3.10)[6]를 사용하여 측정한 결과이다. 추가로 뱅크 충돌로 인한 소요 시간을 정확하게 확인하기 위해 뱅크 충돌이 일어나는 공유 메모리 사용 부분을 제외한 전역 메모리 접근 시간(NoSharedRead)을 나타내었다. ‘warp serialize’ 항목은 뱅크 충돌의 발생 여부를 보여주며, 수치가 높을수록 많은 수의 뱅크 충돌이 발생했음을 보여준다[7].

전역 메모리 접근 시 하프 워프 쓰레드가 읽고자 하는 전체 메모리 영역의 크기를 통해 메모리 트랜잭션이 발생하게 된다. 트랜잭션이란 전역 메모리를 읽어오는 단위 영역을 말하는데 트랜잭션 크기 이외의 메모리 영역을 더 참조해야 하는 쓰레드가 존재할 경우 메모리 트랜잭션 횟수가 증가하게 된다(그림 2).

하프 워프 단위의 쓰레드가 되도록 조밀한 메모리 영역을 참조하고자 할 경우 트랜잭션 횟수는 적게 발생하며, 한 번의 트랜잭션으로 전역 메모리 접근이 완료될 경우 최적의 성능을 발휘하게 된다. CUDA Compute Capability 1.2 이상의 GPU에서 전역 메모리 접근은 하프 워프 단위의 모든 쓰레드가 하나의 같은 세그먼트(segment)를 참조할 경우 한 번의 트랜잭션으로 합쳐진다(coalescing). 또한 전송 되는 메모리 단위는 접근하는 전역 메모리 크기에 따라 결정된다. 트랜잭션이 발생하는 전역 메모리의 단위 영역 시작 주소는 세그먼트의 배수로 결정되며, 최종적으로 128/64/32 바이트 단위 영역을 기준으로 하프 워프가 필요로 하는 최소, 최대 범위의 메모리 주소를 참조하여 최소한의 단위 영역을 결정하게 된다[4].

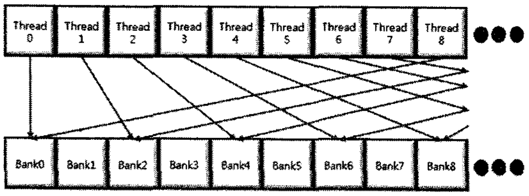


그림 1 두 번의 뱅크 충돌이 발생하는 상황

표 1 뱅크 충돌의 영향(시간 단위: μsec)

	GPU time	occupancy	instructions	warp serialize	gld 64b & gst 64b
BankConflict	2179.21	1	526119	553391	92160
NoBankConflict	1516.24	1	510760	0	92160
NoSharedRead	1503.36	1	510758	0	92160

	NoBankConflict	BankConflict
Shardmemory read time	12.88	675.85

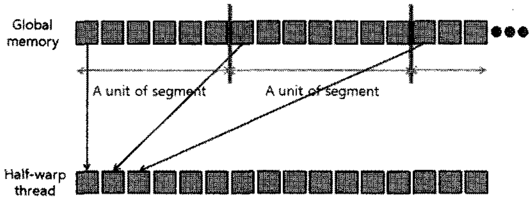


그림 2 좋지 않은 트랜잭션 발생 상황

예를 들어 하프 워프 단위의 모든 쓰레드가 32바이트의 단위 영역 내에서만 메모리를 참조할 경우 32바이트 단위 메모리 트랜잭션이 1번만 발생하게 되며, 참조하고자 하는 전역 메모리가 2,048바이트의 영역 내에 흩어져 있을 경우 128바이트의 트랜잭션이 최대 16번 발생하여 상당한 성능 저하를 초래한다(그림 2).

표 2는 1024×1024개의 부동 소수점 데이터에 대해 128바이트 간격으로 접근하는 상황에 대해서 트랜잭션 크기를 다르게 하여 전역 메모리 접근을 하는 프로그램을 NVIDIA 프로파일러로 측정한 결과이다. Test1 항목은 한 번의 트랜잭션으로 전역 메모리 접근이 가능하도록 하였으며 Test2는 한 번의 트랜잭션으로 가능하지 않도록 구성하였다. Test1에 비해 약 16배의 메모리 트랜잭션이 일어나는 동시에 상당한 성능 저하가 발생함을 확인할 수 있었다.

표 2 트랜잭션의 영향(시간 단위 : μsec)

	GPU time	gld_32b	gld_64b
Transaction Test1	100.80	0	6544
Transaction Test2	2458.34	104960	0

2.3 지역 메모리와 전역 메모리 접근 속도

CUDA API로 프로그래밍 시 레지스터의 사용량이 상당히 많아질 경우 고려할 수 있는 두 가지 방안이 있다. 첫째, SM 당 쓰레드의 활성화 비율을 조금 떨어뜨리더라도 메모리 접근 횟수가 많아 레지스터를 더 사용하는 것이 유리한 경우와 둘째, 컴파일러에서 각 쓰레드 당 사용할 레지스터 수에 제한을 둠으로써 SM 당 쓰레드 활성화 비율을 최대로 하고 지역 메모리를 사용하는 경우이다.

컴파일러에 의해 각 쓰레드 당 설정되는 지역 메모리는 전역 메모리상에 할당되며 전역 메모리의 트랜잭션과 동일한 전송 패턴으로 접근하게 된다. 지역 메모리는 레지스터에 비해 매우 느리기 때문에 만약 지역 메모리 접근이 여러 번 반복해서 일어난다면 활성화 비율을 떨어뜨리더라도 레지스터를 사용하는 것이 더 효과적이다.

다수의 트랜잭션이 발생하게 될 경우 공유 메모리를 버퍼처럼 사용해서 트랜잭션 수를 줄이는 것이 유리하며, 메모리 접근 시 텍스처 유닛을 사용하여 성능 향상을 유도한다.

3. 활성화 비율(Occupancy) 조절 및 지연시간 감춤(Latency hiding)을 통한 최적화

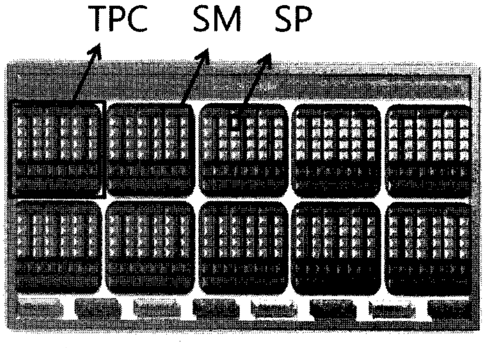


그림 3 NVIDIA GeForce GTX 280 구조[8]

3.1 활성화 비율

활성화 비율은 ‘SM 당 활성화된 워프 수 / SM이 활성 가능한 최대 워프 수’로 정의한다[4]. 활성화 비율을 높여야 하는 이유는 활성화 된 쓰레드들이 많아야 메모리 참조 시 발생하는 지연 시간 동안 다른 활성화된 쓰레드에 대한 연산을 수행하도록 함으로써 SM이 유휴 상태로 빠지게 되는 것을 방지할 수 있기 때문이다. SM 당 활성화된 쓰레드가 적을 경우 메모리 참조 지연 시간동안 SM이 유휴 상태에 빠질 가능성이 높아지고 이는 GPU의 연산 능력을 최대한 활용하지 못함으로 성능 저하를 유발한다.

각 SM에는 자원 즉 레지스터, 쓰레드 수, 공유 메모리가 허용되는 한 최대한의 블록들이 할당되어 활성화되며 한 블록은 한 SM에서만 처리된다. 전체 이용 가능한 자원을 블록들이 나누어 사용하므로 블록 당 사용할 수 있는 자원을 고려하여 공유 메모리와 레지스터 수를 설정해야 한다.

3.2 블록 크기와 활성 블록 수

더 많은 활성 워프 수는 SM를 유효화 하고 지연 감춤 효과를 유도 할 수 있다. 블록 당 쓰레드 수는 워프의 배수 크기로 128 혹은 256개 이상의 쓰레드를 할당하도록 하며, SM 당 최대 블록 수는 제한되어 있으므로 이를 고려하여 블록 크기를 결정한다. 현재 SM 당 최대 활성 블록 수는 8개이다.

하나의 쓰레드가 사용하는 레지스터 수와 SM 당 레지스터 수, 사용하는 공유 메모리의 크기를 고려하여 활성 블록 수를 예측해 볼 수 있다. 활성 블록 수를 고려하여 활성 워프 수가 최대가 되도록 하는 것이 중요하며, 활성 워프 수가 많아지면 블록 내 워프들이 스케줄링 된 후에도 추가 유휴 시간이 발생할 때 다른 블록에 대한 연산을 수행할 수 있어 효율을 더욱 높일 수 있다.

블록 당 최대 512개의 쓰레드를 사용할 수 있으며, CUDA의 Compute Capability 1.2 이상 GPU의 경우 SM 당 최대 활성 쓰레드의 개수는 1024, 레지스터는 16,384이다[4]. 활성화 비율을 최대로 하기 위한 사용 가능 최대 레지스터 수는 ‘SM 당 레지스터 수 16K / (활성 블록 수×블록 당 쓰레드 수)’가 된다. 활성화 비율이 1이 되기 위해서는 사용 레지스터 수가 16 이하가 되어야 한다.

활성 블록 수는 ‘SM 당 최대 활성 쓰레드의 개수 / 블록 당 쓰레드 수’이다. 예를 들어 블록 당 256개의 쓰레드가 쓰인다면 활성 블록 수는 SM에서 최대 4개(1024/256)가 가능하게 되며 SM 당 공유 메모리 크기가 16KB이므로 블록 당 4KB 이하의 공유 메모리 사용이 활성화 비율을 1로 만든다.

3.3 활성 블록 수와 자원

먼저 GPU Compute Capability를 확인하여 SM 당 최대 활성 쓰레드 수를 확인하고, NVIDIA 프로파일러에서 ‘occupancy’ 항목을 확인한다. 컴파일러 옵션 -ptxas-options=-v를 통해 커널이 사용하는 레지스터 수와 지역 메모리, 공유 메모리, 상수 메모리 수를 알

수 있다. 프로그래머의 의도와 달리 레지스터 대신 지역 메모리가 사용되었을 수 있으므로 컴파일 결과를 통해 이를 반드시 확인한다. 또한 레지스터가 의도했던 것 보다 더 많이 사용되어 활성화 비율을 떨어뜨릴 수 있는데 컴파일러 옵션 -maxrregcount를 사용하면 최대 레지스터 수를 제한할 수 있다[9].

GPU 성능을 최대한 활용하기 위해서 활성 워프 수와 블록 수는 자원이 허락하는 한 많을수록 좋다. 블록 수는 최소 SM 수가 되도록 하고, 활성 워프를 많게 하여 쓰레드 동기화나 메모리 지연으로 인한 시간동안 SM이 다른 워프 및 블록을 실행할 수 있도록 한다.

3.4 상대적인 전역 메모리 참조 시간

전역 메모리 참조는 상당한 수의 사이클을 소비하기 때문에, 프로그램 수행 시 참조 횟수에 따라 성능이 크게 변한다. 같은 쓰레드 블록, 그리드 구조의 프로그램은 커널 내부에서 더 적은 횟수의 전역 메모리 참조를 하는 것이 더 빠르다. 여기서의 전역 메모리 참조는 지역 메모리 참조를 포함한다.

그러나 전역 메모리 참조를 하는데 걸리는 시간은 조건에 따라 크게 달라진다. 같은 전역 메모리 참조라도 커널의 전체 실행시간에 끼치는 영향은 더 클 수도 작을 수도 있는데, 커널에서 전역 메모리 참조를 할 때 발생하는 지연 감춤 때문이다[10].

3.5 CUDA의 전역 메모리 지연 감춤

CUDA에서 명령은 32개씩 워프 단위로 실행된다. 하지만 SM 안의 프로세스 코어와 TPC의 전역 메모리 읽기/쓰기 단위 개수의 제한이 있기 때문에, 전역 메모리 참조는 한 블록 내부에서도 동시에 일어나지 않는다. 전역 메모리 참조가 발생할 때 하나의 TPC 스케줄러는 전역 메모리 참조의 긴 시간동안 SM을 쉬게 하는 것이 아니라, SM이 다른 워프를 실행하도록 하여 작업을 계속 하도록 한다. 즉, 전역 메모리 참조가 실행되고 있는 동안에 다른 명령이 실행되며, 이것을 통하여 전역 메모리 참조의 지연 감춤이 일어난다.

따라서 전역 메모리의 지연 감춤은 다음과 같은 조건에서 최적으로 발생할 수 있다.

(1) 충분한 수의 활성 블록 수가 확보되어야 한다.

(2) 커널 내부에서, 하나의 전역 메모리 참조 당 충분한 시간의 명령 수행시간이 확보되어야 한다.

활성 블록 수가 주는 영향은 그림 4에서 나타난다. 블록 하나당 128개과 512개의 쓰레드를 사용할 경우의 NVIDIA GeForce GTX 280의 경우 SM 당 활성 블록 수는 각각 8과 2이다. 8개의 활성 블록 수의 경우가 2개의 활성 블록 수에 비해 수행 시간이 더 안정화 되어 있음을 알 수 있다.

(2)의 경우는 전역 메모리 읽기 및 쓰기 횟수를 변화시키며 측정한 실험 결과인 그림 5, 6과 7에 나타난다. 전역 메모리 읽기와 쓰기는 실행 시간이 비슷하고, 지연 감춤의 효과도 비슷함을 확인 할 수 있다. 덧셈, 곱셈, 나눗셈에 대한 실험결과는 명령 수행 시간 길이가 전역 메모리 참조 시간 길이와 비슷해 질 때까지 전역 메모리 참조 시간만 드러나, 지연 감춤의 효율이 증가함을 보여준다. 명령 수행 시간 길이가 전역 메모리 참조 시간 길이를 초과할 때부터 커널의 수행시간은 실행해야 할 연산이 증가함에 따라 일정하게 증가함을 볼 수 있다.

이러한 지연 감춤 효과는 좀 더 유연한 전역 메모리의 사용을 가능하게 한다. 그림 5에 도시된 결과에 의하면, 이 실험에서 72회 이상의 곱셈 연산에 준하는 명령 수행시간을 가진 커널은 4번 이하의 전역 메모리 참조 시간이 전체 실행 시간에 영향을 주지 않음을 확인할 수 있었다. 더 적은 연산 시간을 가진 커널에서는 전역 메모리 참조 횟수에 따라 커널의 수행시간이 영향을 받게 된다.

3.6 지연 감춤과 __syncthreads()

__syncthreads()는 각 쓰레드의 작업을 모두 동일하게 맞추기 때문에 지연 감춤의 효과를 떨어뜨린다. 이는

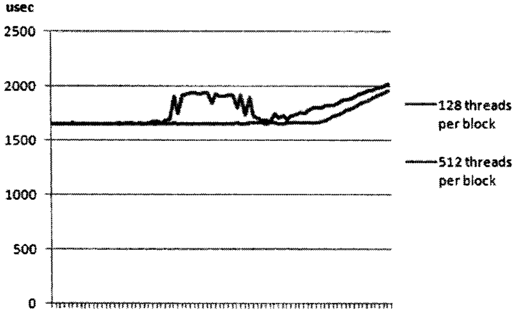


그림 4 곱셈 연산 수와 수행시간, 전역 메모리는 4번 접근, 블록 크기는 각각 32×4, 32×16

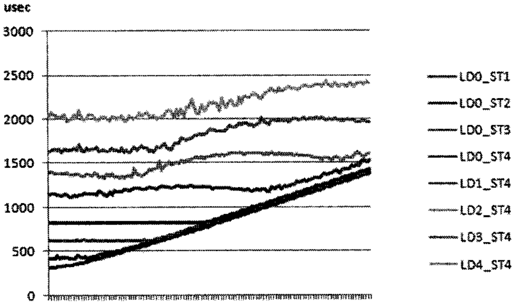


그림 5 곱셈 연산 수와 수행시간
LD/ST : 커널의 전역 메모리 읽기/쓰기 횟수

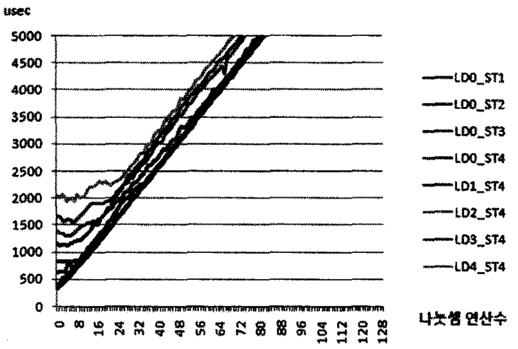


그림 6 나눗셈 연산 수와 수행시간

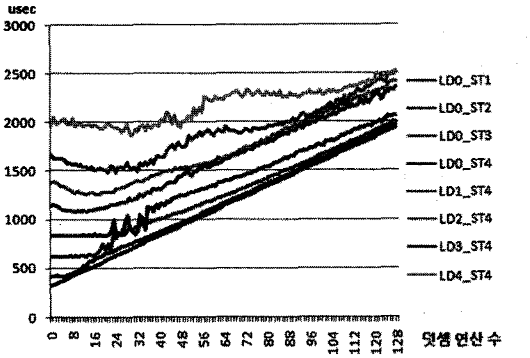


그림 7 덧셈 연산 수와 수행시간

LD/ST : 커널의 전역 메모리 읽기/쓰기 횟수

공유 메모리 사용이 반드시 유리하지 않음을 나타낸다. 공유 메모리에 전역 메모리 일부를 읽어 데이터를 처리하는 방법을 많이 사용하지만, 공유메모리에 전역 메모리를 복사한 후 __syncthreads()를 사용할 경우 지연 감춤은 일어나지 않는다. 3절의 실험 결과는 공유메모리를 사용하지 않는 것이 오히려 전체적인 성능을 향상시킬 수도 있음을 보여준다.

즉, 전역 메모리 참조 횟수 대비 적당한 시간의 명령 수행 시간이 확보 되며, __syncthread()가 그 명령 수행 시간 사이를 구분 짓지 않는 경우, 지연 감춤이 일어나는 전역 메모리만 사용하는 경우가 공유 메모리를 함께 사용하는 것 보다 더 나을 수 있다.

4. 소벨(Sobel) 연산 문제를 통한 최적화 기법의 영향 분석

4.1 소벨 연산을 이용한 경계 검출

소벨 알고리즘은 가로와 세로 두 개의 필터를 기초로 가중치를 계산하여 색의 경계를 추출하는 방법이다[11]. 본 논문의 실험에서 사용한 두 필터와 입력 영상에 대한 결과 이미지는 각각 그림 8, 9와 10에 주어져 있다. 대상이 되는 픽셀을 기준으로 주변 픽셀 데이터에 대해 가로, 세로 필터의 가중치를 곱해 합을 구한 후 각각의 절대 값의 합을 소벨 연산 알고리즘의 결과로 취한다.

CUDA를 사용한 이미지 처리 시에 비활성 쓰레드들을 줄이고 병렬성을 높이기 위한 여러 가지 방법들이 제시된 기존 연구가 있었으며[12], 이미지 처리는 동일한 방법들이 반복됨으로 병렬 처리로 CPU대비 큰 성능 향상을 얻을 수 있는 좋은 예가 된다.

본 논문에서 측정한 소벨 연산 프로그램은 OpenGL 파이프라인을 통해 렌더링한 이미지를 픽셀 버퍼 오브젝트(pixel buffer object)[13]를 이용해서 GPU에 맵핑시켜 경계 추출을 하고, 그 결과를 다시 픽셀 버퍼 오브젝트에 저장 후 텍스쳐 이미지로 바인딩 하여 디스플레이 하는 구조로 되어 있다.

4.2 공유 메모리 기반 최적화 기법

4.2.1 초기 방법

방법 I: 각 픽셀에 대해 소벨 연산적용 시 이전 픽셀을 계산 할 때 참조했던 픽셀을 반복해서 참조해야 한다. 따라서 전역 메모리만을 사용할 경우(부록 [코드 1] 참조) 중첩된 전역 메모리 참조가 발생하므로 접근 속도가 빠른 공유 메모리에 데이터를 옮기는 것이 유리하다. 공유 메모리로 데이터를 복사하고 쓰레드 동기화 함수인

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

그림 8 가로 필터[11]

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

그림 9 세로 필터[11]

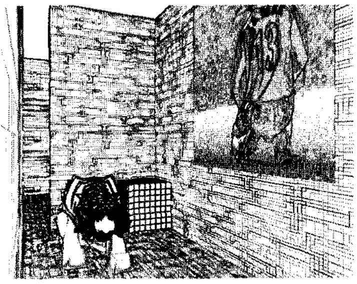


그림 10 소벨 연산 결과 영상

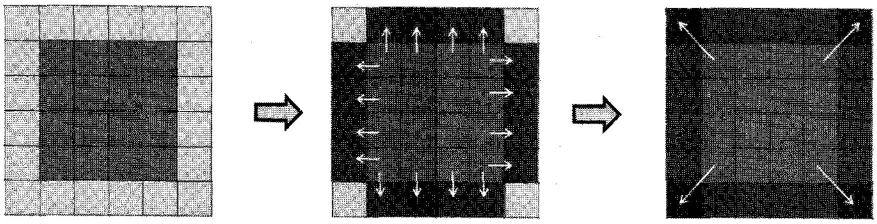


그림 11 if 문을 이용한 공유 메모리로의 복사

__syncthread()를 통해 데이터가 공유 메모리로 완전히 옮겨지게 되면 공유 메모리에서 경계 검출 연산을 한다.

그림 11과 같이 4×4 크기의 블록을 가정하자. 소벨 경계 검출 방법은 대상 픽셀 주위로 픽셀들이 더 필요하므로 공유 메모리는 6×6 크기로 쓰레드 블록 크기보다 크게 할당되어야 한다. 먼저 블록 내 쓰레드들이 전역 메모리의 픽셀 데이터를 공유 메모리에 복사한다. 블록 주변의 아직 복사되지 않은 값들은 블록의 가장자리와 모서리에 있는 쓰레드가 복사한다. 즉, 모든 쓰레드가 한 번씩 복사하며 가장자리에 있는 쓰레드들은 두 번, 모서리에 있는 쓰레드 들은 네 번의 전역 메모리 접근 및 공유 메모리로의 복사를 시행한다(부록 [코드2] 참조).

문제점: 이 방법의 문제점은 표 3에서 볼 수 있듯이 if 문의 사용으로 분기가 일어나는 것이다. 동일한 임의의 이미지에 대해 if 문의 사용으로 분기가 훨씬 많이 일어남을 알 수 있다. 블록 가장자리 쓰레드를 위한 4번의 if 문, 모서리 쓰레드를 위한 4번의 if 문이 필요하게 되어, 총 8번의 if 문이 모든 쓰레드에서 발생한다.

4.2.2 if 문을 제거한 개선 방법

방법Ⅱ: if 문을 전혀 사용하지 않고 블록 크기의 복사 4번으로 공유 메모리를 채우는 방법이다. 그림 12와 같이 왼쪽 위, 오른쪽 위, 왼쪽 아래, 오른쪽 아래 부분의 공유 메모리 값을 채운다(부록 [코드3] 참조).

문제점: 분기문은 없어졌지만 공유 메모리를 채우기 위해 한 쓰레드 당 4번의 전역 메모리 접근이 발생한다. 이는 필요 이상의 메모리 접근 횟수와 공유 메모리 영역 내에서 쓰레드 블록 크기에 해당하는 영역에 대해 같은 데이터를 중복 복사하는 낭비를 초래한다.

4.2.3 중복된 메모리 접근을 제거한 개선 방법

방법Ⅲ: 그림 13은 if 문과 중복된 전역 메모리 접근을 제거하여 공유 메모리를 채우는 과정이다. 각 쓰레드는 공유 메모리 내의 지정된 위치에 전역 메모리의 데이터를 복사한다. 필요한 공유 메모리 크기가 블록 크기보다 크기 때문에 한 번의 복사로 다 채워지지 않으므로 그림 13의 진한 녹색 위치부터 다시 데이터를 채운다. 각 쓰레드는 한 번 또는 두 번의 전역 메모리 접근이 필요하다(부록 [코드4] 참조).

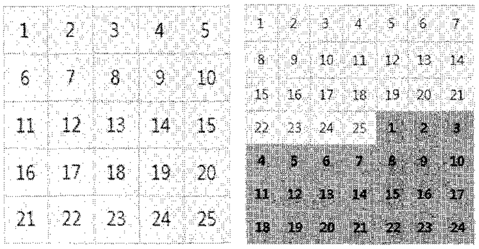


그림 13 중복되지 않는 두 번의 복사. 공유 메모리 블록의 숫자는 쓰레드의 인덱스를 의미

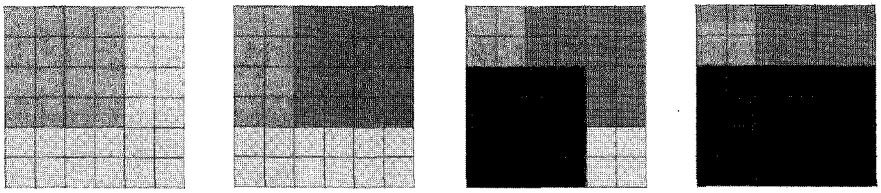


그림 12 블록 크기의 4번 복사

표 3 if 문과 분기 발생 수

	Occupancy	Branch	Divergent branch	Instruction
if문 사용	1	7189	117	64625
Block 4번 사용한 방법	1	2093	13	55418

문제점: 공유 메모리를 채우기 위해 나눗셈과 나머지 연산을 통한 인덱스 계산이 필요하다. 이는 성능의 큰 저하를 가져온다[4].

4.2.4 최종 방법

방법Ⅳ: 블록의 높이를 2로 고정하고, 공유 메모리 높이를 2의 배수 크기로 하는 방법이다. 예를 들어 블록의 크기가 8×2(그림 14), 공유 메모리의 크기가 10×6이라 하자. 그림 15에서와 같이 공유 메모리에 값을 블록 크기씩, '높이 / 2' 번 채운다. 이로써 if 문이나 중복되는 작업 없이 전역 메모리의 값을 공유 메모리에 옮기는 작업이 완성된다. 단, 마지막 2열은 채워지지 않는다.

이러한 공유 메모리에서 경계 검출 작업을 하는 알고리즘을 살펴보자. 그림 16의 초록색과 붉은색으로 표시된 쓰레드 블록크기에서 소벨 연산이 일어난다. 계산이 완료되면 그 아래에서 같은 크기의 영역에 대해 다시 검출이 시작된다. 이 때 붉은색으로 표시된 쓰레드는 공유 메모리상의 데이터가 있지 않은 영역을 참조하는 부분이므로 다른 쓰레드와 똑같이 연산을 수행하지만 최종 결과 값으로는 사용하지 않는다. 이미지 경계 검출을 완성하기 위해 올바른 값이 계산 되지 않는 두 열을 고려하여 다음 블록은 그림 17과 같이 붉은색 영역이 경계 검출 시작 위치가 되도록 점선이 가리키는 부분의 데이터를 전역 메모리에서 공유 메모리로 옮겨오게 된다.

이 방법은 공유 메모리가 부족할 경우, 공유 메모리를 재사용하여 활성화 비율을 높일 수 있는 장점 또한 있다. 예를 들어 10×6 크기의 공유 메모리만을 사용하여 대상 이미지 영역에 대해 계산을 완료한 후 전역 메모리의 그 다음 이미지 영역에 대해 같은 공유 메모리를 재사용하여 다시 소벨 연산을 적용할 수 있다(부록 [코드5] 참조).

문제점: 블록 당 4개 쓰레드는 사용되지 않는 데이터에 대해 연산을 하며 그 결과 값은 버려진다.

4.2.5 실험 결과

표 4는 4.2절의 공유 메모리 최적화 방법들을 적용한 시간 결과이다. 반복적으로 사용되는 필터 값 저장을 위해 상수 메모리를 사용하였고, 본 논문에서는 NVIDIA GeForce GTX 280 GPU에서 4096×3072 이미지에 대해 100회의 소벨 연산 이미지 처리에 대한 평균 수행 시간을 측정하였다. 방법Ⅰ은 공유 메모리를 사용할 때 기본이 되는 모델이다(부록 [코드2] 참조). 방법Ⅱ를 적용하였을 때 중복되는 전역 메모리 읽기가 있음에도 비교문의 제거가 속도를 향상시켰다.

방법Ⅲ은 비교문 및 중복되는 전역 메모리 읽기를 제거하여 속도 향상을 기대했지만 오히려 속도가 저하되었다. 이는 인덱스 계산에 필요한 늘어난 계산 비용이

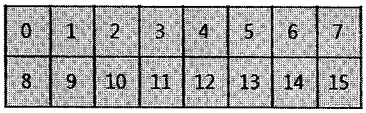


그림 14 8×2의 블록

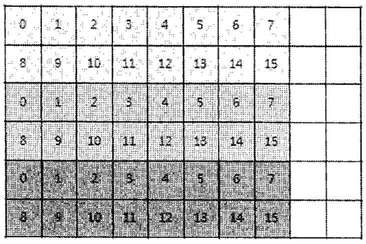


그림 15 공유 메모리에 블록 크기씩 복사

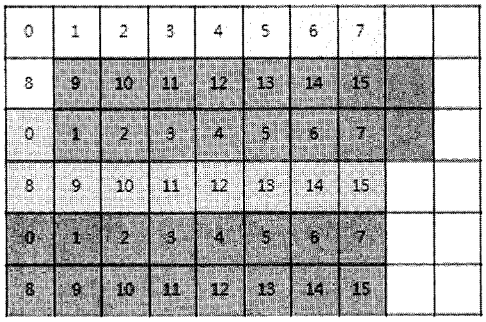


그림 16 하나의 블록 크기에 대한 소벨 연산 경계 검출 영역

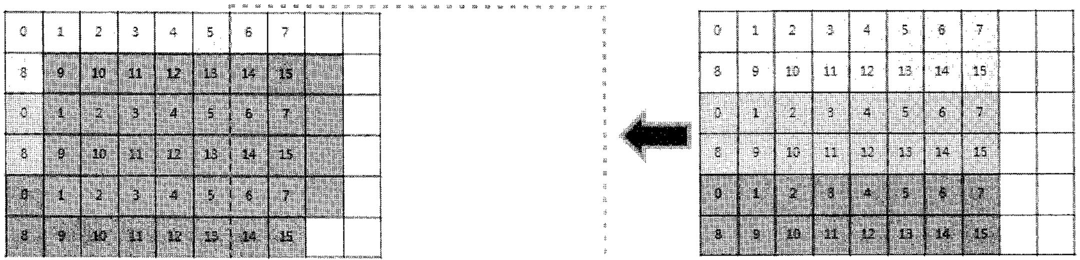


그림 17 다음 공유 메모리에 점선 부분에 해당하는 전역 메모리를 복사

표 4 각각의 최적화 방법에 대한 시간 측정 결과

공유 메모리 최적화	실행 시간(sec)
방법 I	0.015417
방법 II	0.013695
방법 III	0.018475
방법 IV	0.012774

줄어든 트랜잭션 발생량의 효율보다 더 크기 때문이다.

방법IV는 다른 방법보다 전역 메모리 읽기에 의한 트랜잭션 발생량도 적고, 인덱스 계산의 부담도 적으며, 비교문도 사용되지 않는다. 단점은 블록 당 네 개 쓰레드의 픽셀 계산이 버려지는 것인데, 실제 256개, 512개의 큰 블록을 사용할 경우 그 영향은 작아진다.

4.3 기타 최적화 기법 적용 결과

기타 일반 최적화 기법들을 소벨 연산에 적용한 결과를 표 5에 나타내었다. 공유 메모리 사용은 4.2.4절의 방법에 최적화를 적용하였으며 텍스처 메모리 사용은 전역 메모리 사용 코드에서 전역 메모리 읽기를 1차원 텍스처 유닛을 통해 읽도록 하였다.

4.3.1 상수 메모리 사용

상수 메모리 영역은 캐시 가능한 메모리로 캐시 부적중이 발생했을 때 전역 메모리로부터 데이터를 읽으며, 캐시에서 읽는 것은 하프 워프 크기의 쓰레드에 대해 모두 동일한 주소를 접근할 때 레지스터에서 읽는 것만큼 빠르다[4]. 소벨 연산에서 적용되는 필터는 계산마다 매번 접근이 일어나므로 필터를 상수 메모리에 저장하면 한 번만 전역 메모리에서 읽고 이후에는 캐시 적중으로 레지스터만큼 빠른 속도로 참조가 일어난다.

4.3.2 조건문 최적화 및 나눗셈의 곱셈화

CUDA에서 비교문은 워프를 순차적으로 처리하도록 한다. 영상 처리 시 이미지 크기의 범위 제한을 위해 사용하는 비교문을 최소화함으로 분기를 줄인다.

부동 소수점형의 나눗셈 연산은 곱셈 연산에 비해 상당히 느리다[4]. 나눗셈 연산의 사용은 가급적 줄이고 역수를 취해 동등한 곱셈 연산으로 고친다.

4.3.3 변수의 상수화

소벨 연산에서 각 쓰레드가 경계 검출 연산을 할 픽셀의 위치는 블록 인덱스와 쓰레드 인덱스를 통해서 계산된다. 특히 공유 메모리 사용 시 인덱스 계산을 위해 블록 크기를 빈번하게 이용하는데, 블록의 크기는 컴파일 시에 미리 정해지므로 이를 상수로 변환시킬 수 있다. 표 4의 방법III은 'blockDim.x' 및 'blockDim.y' 호출이 다른 방법들에 비해 많다. 이를 상수화 하였을 때 큰 폭의 속도 향상이 있었다.

4.3.4 반복문 제거

소벨 연산 시 가로 세로 주변 9개의 픽셀을 접근하기 위해 두 개의 반복문을 사용하는데 이를 제거한다. 반복문 제거(loop unrolling)는 다른 최적화 단계가 모두 적용 되고 알고리즘이 잘 정비 된 후 마지막에 적용하는 것이 좋다.

4.4 실험 결과

4.2절의 공유 메모리 최적화 기법과 4.3절의 일반적인 최적화 기법을 소벨 연산에 적용한 결과는 최적화를 적용하는데 좋은 참조 자료가 될 것이다.

표 5에서 주목해야 하는 부분은 반복문 제거까지의 최적화가 적용되기 전에는 기대와 달리 공유메모리를 사용한 쪽이 전역 메모리만 사용한 쪽보다 느리다는 것이다. 전역 메모리만 사용할 경우는 소벨 연산 시 한 픽셀 당 9번의 읽기와 1번의 쓰기, 총 10번의 참조를 한다. 공유 메모리 사용 시 픽셀 당 전역 메모리 읽기 1.5번, 공유 메모리 읽기 9번, 전역 메모리 쓰기 1번이 있다. 공유 메모리 읽기는 충분히 빠르다고 할 때, 전역 메모리 참조 횟수 비율이 10:2.5임에도 공유 메모리를 사용한 쪽이 느리다.

이는 3.5절에서 언급한 지연 감춤으로 인한 결과이다. 최적화가 적용된 후에는 반복문 제거 등으로 전역 메모리 참조 횟수 대비 명령 실행 시간이 줄었기 때문에 지연 감춤의 효과가 작아진다. 따라서 전역 메모리의 메모리 접근 시간이 상대적으로 길어지고, 이는 공유 메모리를 사용하는 것이 더 빨라지는 결과로 나타난다.

본 실험 결과에서 텍스처 메모리가 가장 빠른 속도를 보여준다는 점은 소벨 연산의 특성상 밀집된 전역 메모리 참조가 일어나고 이에 따라 높은 캐시 효율이 나타나기 때문이다.

5. 행렬과 벡터의 곱셈을 통한 최적화 기법의 영향 분석

본 절에서는 행렬과 벡터의 곱셈을 통한 최적화 기법

표 5 NVIDIA GeForce GTX 280 GPU에서 4096×3072 이미지에 대한 소벨 연산에 최적화 기법들을 적용한 결과

최적화 기법	공유메모리사용(sec)	전역메모리사용(sec)	텍스처메모리사용(sec)
최적화 없음	0.016354	0.029612	0.025254
상수 메모리 사용	0.012774	0.011710	0.013052
조건문 최적화 및 나눗셈의 곱셈화	0.010962	0.010091	0.011383
변수의 상수화	0.010437	0.009992	0.011732
반복문 제거	0.007262	0.008441	0.006648

적용의 영향을 분석한다. 행렬과 벡터의 곱셈은 2차원 행렬과 1차원 벡터의 곱을 계산하는 문제이다. 입력으로 NxN 사이즈의 행렬 M, Nx1사이즈의 행렬 V가 주어지며, Nx1 행렬 W가 생성된다.

5.1 초기 방법의 알고리즘

각 그리드와 블록은 1차원으로 선언된다. 커널에서 하나의 쓰레드는 결과 벡터 W의 복수의 항에 들어갈 값을 계산한다. 행렬 M의 블록 ID번째 행과 벡터 V의 벡터내적을 통하여 벡터 W의 블록 ID번째의 항을 계산하며, 쓰레드 크기보다 큰 행은, '행 번호 mod 쓰레드 크기'번째의 쓰레드가 계산을 한다.

5.2 최적화된 방법의 알고리즘

하나의 쓰레드가 행렬 M의 한 행을 계산하도록 하기보다 하나의 블록이 행렬 M의 한 행에 해당하는 곱셈을 계산하도록 하였으며 전역 메모리 접근 시 최소의 트랜잭션이 일어나도록 하였다.

1차원의 그리드와 블록에 대해 블록의 각 쓰레드들은 M의 '열 번호 mod 블록 크기' 번째 열과, V의 '행 번호 mod 블록 크기'번째 행의 곱을 모두 더한다(그림 18). 그 후 각각의 쓰레드가 계산한 값을 취합하면 결과 W의 한 항의 값이 생성된다.

취합 과정은 전체 쓰레드 블록이 1/2씩 공유 메모리 크기를 줄여가며 그 값을 누적해 나가는 방식을 사용한다(부록 [코드6] 참조).

5.2.1 블록크기와 활성 블록 최적화

최고의 활성화 비율 및 쓰레드 스케줄링 효과를 낼 수 있도록 쓰레드 블록 크기를 블록 당 최대 쓰레드 수, 그리드 크기를 SM의 수 × 2로 하였다. 본 논문 실험환경에서 최대 블록 크기는 512가 되며, 활성화 비율을 고려하여 SM 당 2개의 블록이 할당되도록 하여 그리드의 크기를 60으로 하였다.

5.2.2 메모리 최적화

초기방법이 커널의 입력 데이터인 행렬 M과 벡터 V의 인수 저장에 전역 메모리를 사용한 것에 반해서, 빈번히 접근하는 벡터 V를 텍스처 메모리를 통해 접근하여 상당한 성능 향상을 이루었다. 행렬 M의 경우 커널 내부에 중복된 참조가 없으므로 텍스처 메모리의 사용은 오히려 성능 저하를 가져왔기 때문에 전역 메모리를 사용하였다.

예제에서 한 행에 대한 곱을 위해 '블록 당 쓰레드 개수×부동 소수점 데이터 크기'의 공유메모리를 사용하였다. 공유 메모리의 합을 구하는 과정에서 블록 내 쓰레드가 같은 뱅크에 중복 접근하여 뱅크 충돌이 발생하지 않도록 공유 메모리를 참조하였다.

5.2.3 분기문과 _syncthreads() 제거

행렬과 벡터의 곱셈 후 블록 크기의 공유 메모리에 저장되어 있는 값을 합하는 과정에서 각각의 쓰레드 인덱스를 제어하여 필요한 값만을 합할 수 있다. 그러나 if문을 이용해 쓰레드들을 제어하는 것보다 사용하게 되지 않는 값의 합까지 동일하게 병렬적으로 계산하도록 함으로 성능이 더 향상되었으므로 if문을 제거하였다. 또한 하프 워프 단위 이하 쓰레드의 공유 메모리 접근 시 읽기 및 쓰기 충돌로 인한 오류 값이 발생하지 않기 때문에, 동기화를 위한 __syncthreads() 함수 호출은 필요 없게 된다.

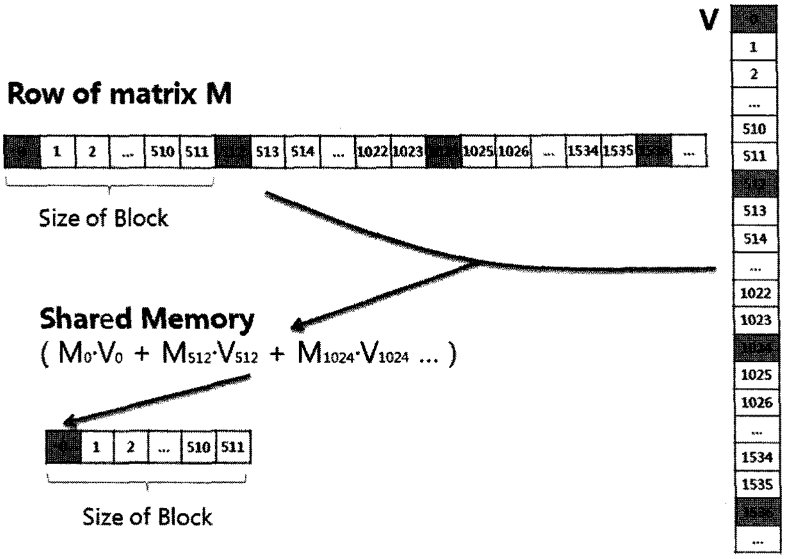


그림 18 블록 내 0번 쓰레드가 수행하는 연산

표 6 행렬과 벡터 곱셈에 대한 최적화 결과

종류	시간
기본 알고리즘	30163.2 μs
최적화 된 알고리즘	558.4 μs

표 6에 최종 결과를 나타내었는데 최적화가 적용되기 전에 비하여 약 56배 빨라진 결과를 확인 할 수 있다.

6. 결 론

본 논문을 통해 CUDA API를 사용한 GPU 프로그래밍에 필수적으로 고려해야 하는 사항들을 살펴보았다. 가장 일반적인 행렬과 벡터의 곱셈 프로그램과 메모리 접근 패턴이 CUDA 구조에서 이상적으로 잘 지켜지지 않는 소벨 연산을 예로 최적화 기법과 그 효과들을 명시적으로 보여주었다. 특히 CUDA 프로그램에 필수적인 메모리 사용에 있어 다양한 경우에 효과적인 메모리 사용 방법을 선택할 수 있는 지표를 제시하였다. 메모리 읽기와 쓰기의 참조 방식에 있어 프로그램의 유형은 다음과 같이 분류할 수 있다.

(1) 다수의 트랜잭션으로 전역 메모리 읽기, 쓰기를 하는 프로그램

(2) 소수의 트랜잭션으로 전역 메모리 읽기, 쓰기를 하는 프로그램

(3) 소수의 트랜잭션으로 전역 메모리 읽기, 쓰기를 하며, 지연 감춤을 기대할 수 있을 정도로 많은 계산을 갖는 프로그램

커널을 최적화하기 위해서는 커널의 알고리즘을 잘 정립하여 읽기와 쓰기 참조 횟수가 최소가 되도록 한다. (1)의 경우 메모리 접근이 빈번할 때에는 공유 메모리를 사용하여 읽기, 쓰기 연산을 하는 것이 좋다. NVIDIA GPU Computing Software Development Kit의 'transposeNew' 프로그램이 좋은 사례이다. (2)의 경우도 공유 메모리 사용을 권한다. 그러나 공유메모리를 사용하는 것이 전역 메모리 참조 횟수를 줄여 이득을 얻지 못하는 경우에는 공유 메모리 사용이 속도 향상을 주지 못한다. (3)의 경우 지연 감춤이 기대될 경우에는 공유 메모리를 사용하지 않고 전역 메모리 혹은 텍스처 메모리를 직접 사용하는 것이 더욱 속도가 빠를 수 있다. (1), (2), (3) 모두에 대하여, 전역 메모리 읽기가 많을 경우 텍스처 메모리의 사용을 권장하며 활성 워프 수가 많도록 한다.

마지막으로 본 논문에서 언급한 요소들이 향후 더 효율적이고 최적화된 CUDA API를 사용한 프로그램을 만드는데 기여할 수 있을 것이라 기대한다.

참 고 문 헌

[1] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and kevin Skadron, *A Performance Study of General-Purpose Applicaions on Graphics Processors Using CUDA*, Journal of Parallel and Distributed Computing, University of Virginia, 2008.

[2] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu, *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.

[3] NVIDIA. http://www.nvidia.com/object/product_geforce_gtx_280_us.html, 2009.

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

[5] Maryam Moazeni, Alex Bui, and Majid Sarrafzadeh, *A Memory Optimization Technique for Software-Managed Scratchpad Memory in GPUs*, University of California, 2009.

[6] NVIDIA. *NVIDIA CUDA Visual Profiler (Version 2.3)*, 2009.

[7] Joe Stam, *Convolution Soup*, NVIDIA, 2009.

[8] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture: Technical Brief NVIDIA GeForce GTX 200 GPU Architectural Overview*, 2008.

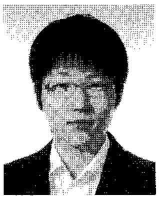
[9] NVIDIA. *Optimizing CUDA*, 2009.

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

[11] Sobel, I., Feldman,G., *A 3x3 Isotropic Gradient Operator for Image Processing*, presented at a talk at the Stanford Artificial Project, 1968.

[12] Victor Podlozhnyuk, *Image Convolution with CUDA*, NVIDIA CUDA 2.0 SDK document, 2007.

[13] Mark Segal, Kurt Akeley, *The OpenGL Graphics System: A Specification(Version 2.1 - December 1)*, 2006.

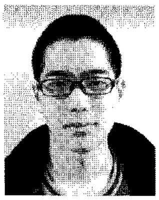


김 성 수

2009년 2월 서강대학교 컴퓨터공학과 졸업(공학사). 2009년 3월~현재 서강대학교 대학원 컴퓨터공학과 석사과정. 관심분야는 컴퓨터 그래픽스, 실시간 렌더링

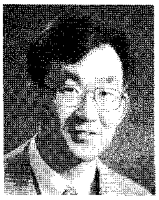
김 동 헌

2009년 8월 서강대학교 컴퓨터공학과 졸업(공학사). 2009년 9월~현재 서강대학교 대학원 컴퓨터공학과 석사과정. 관심분야는 컴퓨터 그래픽스, 실시간 렌더링, 극사실적 렌더링, GPGPU 컴퓨팅



우 상 규

2009년 8월 서강대학교 컴퓨터공학과 졸업(공학사). 2010년 3월~현재 서강대학교 대학원 컴퓨터공학과 석사과정. 관심분야는 컴퓨터 그래픽스, 실시간 렌더링, 극사실적 렌더링, 고성능 계산



임 인 성

1985년 2월 서울대학교 자연과학대학 계산통계학과 졸업(이학사). 1987년 5월 Rutgers - The State University of New Jersey 컴퓨터학과 졸업(이학석사) 1991년 7월 Purdue University 컴퓨터학과 졸업(이학박사). 1999년 7월~2000년 7월 University of Texas at Austin의 TICAM의 연구교수. 1993년 3월~현재 서강대학교 컴퓨터공학과 교수. 관심분야는 컴퓨터 그래픽스, 과학적 가시화, 고성능 계산

부록: 본 논문에서 사용한 CUDA 예제 코드

공유 메모리를 SMEM, 전역 메모리를 GMEM으로 표기함

[코드 1] 전역 메모리만 사용

```
// 가로 세로 필터는 상수 메모리 사용
row = Bid.x*Bdm.x+Tid.x;
col = Bid.y*Bdm.y+Tid.y;
// 해당 픽셀 및 주변 8픽셀 값에 소벨 연산 적용
for 필터의 각 열 y do      //세로
  for 필터의 각 행 x do     //가로
    int pixel = GMEM(row+x,col+y)
    float res = pixel 값 흑백 이미지 변환
    sx += res*가로 필터(x,y)
    sy += res*세로 필터(x,y)
  end for
end for
sx와 sy의 절대값의 합을 전역 메모리에 저장
```

[코드 2] if 문을 사용하여 전역 메모리의 픽셀 값을 공유 메모리에 복사

```
__shared__ SMEM[(Bdm.x+2)*(Bdm.y+2)]
x = Bid.x*Bdm.x+Tid.x;
y = Bid.y*Bdm.y+Tid.y;
// 쓰레드 하나씩 공유 메모리에 복사
SMEM(Tid.x+1 + (Tid.y+1)*(Bdm.x+2)) = GMEM(x,y)
if (Tid.x==0) //left
     SMEM(0,Tid.y+1) = GMEM(x-1,y)
if (Tid.x==Bdm.x-1) //right
     SMEM(Bdm.x+1,Tid.y+1)= GMEM(x+1,y)
if (Tid.y==0) //top
     SMEM(Tid.x+1,0) = GMEM(x,y-1)
if (Tid.y==Bdm.y-1) //bottom
     SMEM(Tid.x+1,Bdm.y+1)=GMEM(x,y+1)
if (Tid.x==0 && Tid.y==0) //top left
     SMEM[0] = GMEM(x-1,y-1);
if (Tid.y==0 && Tid.x== Bdm.x-1) //top right
     SMEM[Bdm.x+1] = GMEM(x+1,y-1)
if (Tid.y==Bdm.y-1 && Tid.x==0) // bottom left
     SMEM(0,Bdm.y+1) = GMEM(x-1,y+1);
__syncthreads();
// 해당 픽셀 및 주변 8픽셀 값에 소벨 연산적용
for() do      // 세로 필터 연산
  for() do    // 가로 필터 연산
  end for
end for
```

[코드 3] if 문을 사용하지 않고 블록 크기의 복사 4번으로 공유 메모리에 채움

```
x = Bid.x*Bdm.x+Tid.x;
y = Bid.y*Bdm.y+Tid.y;
// left top
SMEM(Tid.x,(Tid.y)*(Bdm.x+2))=GMEM(x-1,y-1)
// left down
SMEM(Tid.x,(Tid.y+2)*(Bdm.x+2))=GMEM(x-1,y+1)
// right top
SMEM(Tid.x+2,(Tid.y)*(Bdm.x+2))=GMEM(x+1,y-1)
// right down
SMEM(Tid.x+2,(Tid.y+2)*(Bdm.x+2))=GMEM(x+1,y+1)
__syncthreads();
// 해당 픽셀 및 주변 8픽셀 값에 소벨 연산적용
```

[코드 4] 중복된 전역 메모리 접근을 제거한 방법

```
x = Bid.x*Bdm.x+Tid.x;
y = Bid.y*Bdm.y+Tid.y;
//첫 번째
SMEM(Tid.x,Tid.y*(Bdm.x)) =
GMEM(x+(Tid.y*Bdm.x+Tid.x)%(Bdm.x+2),
        y+(Tid.y*Bdm.x+Tid.x)/(Bdm.x+2))
//두 번째
SMEM(Tid.x, Tid.y*(Bdm.x)+(Bdm.x)*(Bdm.x)) =
GMEM(x+((Bdm.x*Bdm.x)+Tid.y*Bdm.x+Tid.x)%(Bdm.x+2),
        y+((Bdm.x*Bdm.x)+Tid.y*Bdm.x+Tid.x)/(Bdm.x+2))
__syncthreads();
// 해당 픽셀 및 주변 8픽셀 값에 소벨 연산적용
```

[코드 5] 블록의 세로 크기를 2로 고정한 방법

```
x = Bid.x*(Bdm.x-2)+Tid.x;
y = Bid.y*4+Tid.y;
// 6행 선언. 이미지 4행의 픽셀 데이터에 소벨 연산적용이 가능
__shared__ int SMEM[(Bdm.x+2)*6];
```

```
// 공유 메모리에 6행을 채움
SMEM(Tid.x, Tid.y *(Bdm.x+2))=GMEM(x,y-1)
SMEM(Tid.x,(Tid.y+2)*(Bdm.x+2))=GMEM(x,y+1) .
SMEM(Tid.x,(Tid.y+4)*(Bdm.x+2))=GMEM(x,y+3)
__syncthreads();
// 해당 픽셀 및 주변 8픽셀 값에 소벨 연산적용
for() do     // 세로 필터 연산
  for() do   // 가로 필터 연산
  end for
end for
// 마지막 2열을 제외하고 저장
if(Tid.x < Bdm.x-2) {
    GMEM(x,y) = result
}
```

[코드 6] 최적화된 행렬 곱셈

```
threadID = Tid.x; blockID = Bdm.x;
__shared__ float temp[numThreadsPerBlock];

// 각 쓰레드는 필요한 행의 계산을 한다.
for(int y = blockID ; y < M의 세로행 크기 ; y+= 그리드 크기)
{
     for (int x = threadID; x < M의 가로열 크기; x+=
     쓰레드 블록 크기)
           result += M의(x,y) 항 * V의 x항;
     temp[threadID] = result;
     __syncthreads();

     // 공유 메모리에 저장된 값을 모음.
     temp[threadID]+=temp[threadID+8];
     temp[threadID]+=temp[threadID+4];
     temp[threadID]+=temp[threadID+2];
     if(32이하의 threadID에 대해)
        temp[ 512 + (threadID>>4) ] = temp[threadID]
        + temp[threadID+1];
     __syncthreads();
     if(16이하의 threadID에 대해) {
        temp[threadID] = temp[512 + threadID] +
        temp[528 + threadID];
        //__syncthreads()와 특정 threadID를 제한하는
         if문이
        // halfwarp 이하의 단위에서 실행되기 때문에
         필요하지 않다.
      temp[threadID] = temp[threadID] + temp[threadID+8];
      temp[threadID] = temp[threadID] + temp[threadID+4];
      temp[threadID] = temp[threadID] + temp[threadID+2];
      temp[threadID] = temp[threadID] + temp[threadID+1];
      W의 y항 = temp[0];      //결과 저장
   }
  __syncthreads();
 }
```