2013년 4월 전자공학회 논문지 제 50 권 제 4 호
Journal of The Institute of Electronics Engineers of Korea Vol. 50, NO. 4, April 2013 http://dx.doi.org/10.5573/ieek.2013.50.4.108

논문 2012-50-4-14

# 대규모 병렬 시스템에서 캐시와 공유메모리를 이용한 유한 차분법 성능

## ( Performance of the Finite Difference Method Using Cache and Shared Memory for Massively Parallel Systems )

김 현 규*, 이 효 종**

( Hyun Kyu Kim and Hyo Jong Lee )

요 약

최근 GPU 시스템과 같은 수백 개의 프로세서로 구성된 대규모 병렬 시스템을 이용하여 성능을 향상시키는 방법들이 많이 개발 되었다. 대표적으로 GPU에서 캐싱(Caching)과 유사한 개념으로 공유 메모리가 사용되었다. 출력 값을 얻기 위해서 이웃 값을 참조하는 이미지 필터와 같은 알고리즘들의 경우 이웃 값의 참조가 빈번하게 발생되므로 공유 메모리를 사용할 경우 성능이 향상되었다. 그러나 공유 메모리를 사용하기 위해서는 기존 코드를 재 구현해야만 하고 이는 코드의 복잡도를 증가시키는 원인이 된다. 최근 GPU 시스템에서는 공유 메모리 뿐 아니라 L1과 L2 캐시 메모리를 지원하도록 하였다. L1 캐시 메모리는 공유 메모리와 동일한 하드웨어에 위치하여 캐시의 사용이 성능향상을 도와줄 것으로 예측된다. 따라서 본 논문에서는 캐시 메모리와 공유 메모리의 성능을 비교하였다. 연구결과 성능 면에서 캐시 메모리를 사용한 알고리즘과 공유메모리를 사용한 알고리즘은 유사하였다. 특히 캐시 메모리를 사용하는 경우 공유메모리 사용 프로그래밍에서 나타나는 코드 복잡도의 증가 문제도 동시에 해결할 수 있었다.

Abstract

Many algorithms have been introduced to improve performance by using massively parallel systems, which consist of several hundreds of processors. A typical example is a GPU system of many processors which uses shared memory. In the case of image filtering algorithms, which make references to neighboring points, the shared memory helps improve performance by frequently accessing adjacent pixels. However, using shared memory requires rewriting the existing codes and consequently results in complexity of the codes. Recent GPU systems support both L1 and L2 cache along with shared memory. Since the L1 cache memory is located in the same area as the shared memory, the improvement of performance is predictable by using the cache memory. In this paper, the performance of cache and shared memory were compared. In conclusion, the performance of cache-based algorithm is very similar to the one of shared memory. The complexity of the code appearing in a shared memory system, however, is resolved with the cache-based algorithm..

**Keywords** : Anisodtropic diffution filter, CUDA, GPGPU, parallel processor

* 학생회원, 전북대학교 컴퓨터공학부
(Div. of Computer Science and Engineering, Chonbuk National University)
** 평생회원, 전북대학교 컴퓨터공학부, 영상정보신기술연구센터
(Div. of Computer Science and Engineering, CAIIT, Chonbuk National University)
※ 이 논문은 2012년도 정부(교육과학기술부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임
(No. 2012R1A2A2A03).
접수일자: 2013년1월31일, 수정완료일: 2013년3월18일

## Ⅰ. 서 론

미분방정식은 자연 현상에 대한 수치적 풀이에 사용된다. 하지만 미분 방정식의 범위는 무한대를 가지므로 유한한 연산장치를 이용하여 해를 얻을 수가 없다. 이러한 문제로 유한한 공간에서 근사치를 구하여 사용하게 되는데 이를 수치해석이라고 하고 미분방정식에 대한 수치적 해법은 차분 방정식이라고 한다.

차분 방정식에서 현재 차분 값을 계산하기 위해 이웃 점들을 참조하게 되는데 이 때 잦은 메모리 접근으로 인하여 실행 성능은 감소된다. CPU에서는 계산하는 영역을 작은 타일 영역으로 나누고 캐시 메모리에 적재시킨 후 해당 연산을 수행하는 방법으로 성능을 향상시켰다[1]. 또한 주어진 방정식이 계산 집약적인 경우 CPU 수를 증가 시켜 성능 향상을 시킬 수 있다. 최근에는 한 개의 물리적 칩셋에 2 ~ 8개의 코어를 가진 CPU가 등장하였고, 수십 개의 코어로 구성된 병렬 시스템이 곧 출시될 예정이다[2]. 하지만 CPU의 코어 수를 늘리는 것은 물리적 제한이 따른다. 이러한 CPU의 단점을 해결하고자 할 때 GPU 시스템의 활용이 대안이 될 수 있다. 연산 성능을 높이기 위해 GPU는 여러 분야에서 다양하게 활용되고 있다[21-23].

그래픽 전용 연산을 수행하는 GPU시스템은 CUDA (Compute Unified Device Architecture)[3], OpenCL (Open Computing Language)[4], C++AMP (C++ Accelerated Massive Parallelism)[5] 과 같은 언어를 이용하여 GPGPU (General-Purpose computing on Graphics Processing Units)[6]로 수행이 가능하다. GPU 시스템은 수백 개 이상의 프로세서를 가지고 있으며 SIMD (Single Instruction Multiple Data)와 유사한 방법으로 데이터를 처리를 한다. 때문에 2차원 및 3차원으로 구성된 대용량 데이터를 처리하는데 적합하다.

하지만 많은 프로세서가 하나의 전역 메모리에 동시 접근하기 위해 경합을 함으로써 메모리 접근 지연으로 인한 성능 감소가 발생한다[7]. 따라서 대규모 병렬 시스템에서는 메모리 접근 횟수를 줄이는 것이 성능 향상에 중요한 기술이 된다. 이것은 대규모 병렬 시스템이 가지는 구조적인 문제이며 알고리즘 개발단계에서 신중하게 다루어야 한다.

이 문제점을 해결하기 위해서 이전 GPU 시스템에서는 공유메모리만을 사용할 수 있었다. Mickikevicius는 3차원 차분 방정식에서 공유메모리와 레지스터를 활용하는 방법을 제안하였다[8]. 그러나 제안방법은 공유 메모리 사용으로 알고리즘의 구현 복잡도가 증가하였고, 그림1과 같은 십자 구조만을 사용하였기에 더 많은 이웃 점들을 활용하는 개선된 차분 방정식에서는 레지스터들의 제약으로 적합하지 않다. Datta등 도 역시 CPU에서 사용된 방법을 GPU 시스템에 적용하여 성능 향상을 보였으나 공유 메모리를 사용하는 경우 코드의 구현 측면에서 복잡도가 증가 되는 것, 즉 추가적으로 코드가 삽입되는 문제를 피할 수 없었다[9].

공유 메모리를 사용하여 수행 성능을 올리는 방법은 공유메모리만 지원되던 구조에서 발생된 문제이다. CUDA에서는 페르미 아키텍처(Fermi Architecture) 이후부터 캐시 메모리를 지원하고 있으며, CPU에서처럼 전역 메모리로부터 읽어온 값을 캐싱 한다. 캐시 메모리와 공유 메모리는 같은 하드웨어에 위치하고 있으며 실행 환경에 따라서 메모리 공간들이 구분이 된다[10].

그러므로 캐시 메모리를 활용한다면 공유 메모리를 사용하지 않아도 동일한 성능을 볼 수 있을 것이며 공유 메모리 사용으로 나타나는 구현 복잡 도는 피할 수 있을 것이다. 이러한 발전에도 불구하고 지금까지도 GPU 시스템을 활용할 때에 공유 메모리를 활용하여 성능의 최적화를 시도하고 있다. 본 논문에서는 차분 방정식을 활용하는 비등방성 확산 필터를 공유 메모리와 캐시 메모리를 사용하여 구현하고 각각의 성능을 비교하였다.

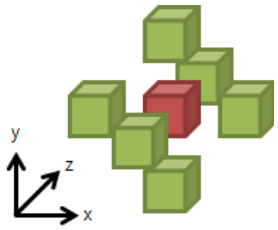



그림 1 3차원 데이터에서 참조되는 점 (적색)과 이웃 점 (녹색)

Fig. 1. A referenced point (red colored) and neighbor points (green colored) in 3-D data

## Ⅱ. 본 론

### 1. 유한 차분법(Finite Difference Method)

유한 차분법은 주어진 미분 방정식을 이용하여 일정

한 간격으로 나눈 다음 테일러급수와 같은 급수 해를 이용한 풀이 방법이다. 따라서 연산장치를 이용한 미분 방정식의 근사 해를 구할 수 있다[11]. 특히 자연현상에서 발생되는 물리적 현상 중 열전도 방정식을 유한 차분법으로 표현하면 식 (1)과 같이 나타낼 수 있다[12].

$$D_{x,y,z}^{t+1} = c_0 D_{x,y,z}^{t} + \sum_{i=1}^{k/2} c_i \begin{pmatrix} D_{x-i,y,z}^{t} + D_{x+i,y,z}^{t} + \\ D_{x,y-i,z}^{t} + D_{x,y+i,z}^{t} + \\ D_{x,y,z-i}^{t} + D_{x,y,z+i}^{t} \end{pmatrix} \quad (1)$$

식 (1)에서 t는 시간축 상에서의 값을 나타내며, 따라서 $D^t$는 현재 데이터이며 $D^{t+1}$은 다음 단계의 값을 나타낸다. 상수 c는 참조하는 이웃 점 거리 i에 따른 적용 계수이다. 단위 거리라고 가정할 경우 식 (1)에 관여하고 있는 데이터 점들을 그림 1에 나타내었다.

### 가. 비등방성 확산 필터에서 유한 차분법

비등방성 확산 필터는 한 점에서 시간이 흐름에 따라 열이 주변으로 확산되어가는 물리적 현상을 이용한 것이다. 이는 열전도 방정식을 이용하여 물리적 현상을 수식으로 기술 할 수 있고 유한 차분법을 적용하면 근사 해를 구할 수 있다. 영상에서 열을 잡음으로 간주하면 시간이 지남에 따라 열이 주변의 온도에 수렴되듯이 잡음이 주변 값에 수렴되어 잡음이 제거되는 원리이다.

이렇게 적용된 결과는 경계 값과 같은 중요한 영상정보는 보존하면서 잡음을 제거할 수 있기 때문에 의료영상과 같이 민감한 영상의 잡음을 제거하는데 유용하게 쓰인다. 입력 영상의 특성에 따라 다르지만 원하는 결과 값을 얻기 위해서 보통 10~100회 정도 반복한다.

비등방성 확산 필터는 대표적으로 기울기를 이용하는 방법과 곡률을 이용하는 방법이 있다. 개선된 비등방성 확산 필터에는 이 두 가지 방법을 기초로 하고 있다[13-17]. 여기에서는 기울기 기반 방식을 사용하였다.

Perona와 Malik[13]에 의해 개발된 기울기를 이용한 비등방성 확산 필터는 식 (2)와 같이 정의된다.

$$f_t = \nabla \cdot c(|\nabla f|)\nabla f \quad (2)$$

여기서 $f = f(x,y,z)$이며 $f(x,y,0) = I(x,y)$는 2차원 입력 영상을 나타낸다. 함수 c는 전도 함수이며 식 3와 같이 정의된다.

비등방성 확산에서 가장 중요한 부분은 전도함수를 정의하는 것이다. 전도 함수는 영상에서 잡음에 따라 다양하게 정의 할 수 있다. 다른 여러 논문들에서 제안된 비등방성 확산은 이러한 전도 함수를 처리 속도나 품질을 개선 한 것이다. 여기서는 초기에 제안된 가우시안 함수를 활용한 방법을 이용하여 비등방성의 성능을 비교하였다.

$$c(|\nabla f|) = e^{-\left(\frac{|\nabla f|}{\kappa}\right)^2} \quad (3)$$

식 (3)는 κ 값 근처에서만 확산이 나타남을 알 수 있다. 델(▽) 연산자는 3차원 영상에서 일반적으로 6개의 이웃 화소들을 참조하지만, 높은 정확도를 위해서 다른 논문들에서 제안한 것처럼 더 많은 이웃 화소들을 참조할 수 도 있다[18].

## 2. CUDA 구조

CUDA에서는 커널 함수를 호출함으로써 작업을 수행한다. 커널 함수는 GPU실행되는 함수를 의미하며 블록(block)과 그리드(grid)로 구성된다. 쓰레드의 집합을 블록이라 부르고, 블록의 집합을 그리드라고 부른다[3].

커널 함수를 실행시키는 CUDA 프로세서는 8의 배수를 하나의 단위로 하여 스트림 멀티프로세서를 구성한다. 페르미 구조에서 하나의 스트림 멀티프로세서는 32개의 CUDA 프로세서를 가지고 있다. 또한 하나의 스트림 멀티프로세서가 실행 시킬 수 있는 쓰레드 수는 1,536개까지 구성할 수 있다[10]. CUDA의 경우 스트림 멀티프로세서를 여러 개로 구성시켜 대규모 병렬 처리가 가능하다. 실제 CUDA를 활용할 때에는 단위 블록당 쓰레드 수의 구성에 따라 성능이 다르게 나타나므로 다양한 구성을 실험하여 최적의 성능을 알아내야 한다.

## 3. CUDA에서 메모리 병합

GPU는 수백 개 이상의 코어를 가지고 있어서 CPU보다 메모리 병목현상이 심각하게 발생한다. 이는 GPU 시스템에서 성능을 감소시키는 중요한 요소이다. 이 문제를 해결하기 위해 CUDA에서는 두 가지 기술을 제공한다.

첫째는 온칩(On-chip) 공유 메모리를 제공한다. 페르미 아키텍처부터 캐시 메모리와 함께 제공한다. 전역 메모리 접근에 비해 10배 이상 빠르게 처리 할 수 있다.

둘째는 오프칩(Off-chip) 메모리 접근 시 병합 기술을 제공한다. 오프칩 메모리에는 전역 메모리가 있으며


그림 2 (a) 일치된 메모리 병합과 (b) 일치되지 않은 메모리 병합
Fig. 2. (a) Coalescences of aligned memory (b) Coalescences of unaligned memory

병합 기술을 이용한다. 병합 기술이란 그림 2(a)와 같이 각 쓰레드가 연속된 메모리 값을 참조 할 경우 한 번 전역 메모리에 접근함으로써 반입(Fetch)을 처리 하는 기술이다. 이를 일치된 메모리 (aligned memory) 접근이라고 한다. 그림 2(b)와 같은 방법으로 접근하게 되면 128바이트크기로 전역 메모리에 2번 접근하게 되는데[19], 이를 불일치 메모리 (unaligned memory) 접근이라 한다.

CUDA에서는 최대 128Byte의 대역폭을 가진다. 이러한 기능은 식 (1)에서 y축과 z축 차분 값을 계산 할 때 이웃 점 접근은 일치된 메모리 접근이 가능하므로 메모리 병합을 이용하여 한번 접근으로 해결 할 수 있다. 하지만 x축의 경우 그림 2(b) 와 같은 일치되지 않는 메모리 접근이 발생 하므로 한번 접근으로 해결 할 수 없다. 이러한 문제는 성능을 감소시키는 요인이 된다. 이와 유사한 현상은 CPU에서도 발생하게 되는데 CPU의 경우 동시에 실행되는 쓰레드 수가 적고 캐시 메모리의 크기가 크기 때문에 GPU에서 만큼의 성능 감소를 일으키지 않는다.

### 4. 쓰레드 구성 및 메모리 병합

2차 이상의 유한 차분 방정식에서 대규모 병렬 시스템을 활용 할 때 쓰레드 구성을 그림 3에서와 같이 일반적으로 사용되는 타일 구조로 구성 할 수 있다. 그림 3에서 사각형은 하나의 쓰레드 집합을 의미한다. 이 방식은 2차원 데이터를 작은 2차원 평면으로 나누어서 처리하는 방식이다. 이때 작은 2차원을 타일이라 부른다. 이 개념은 3차원으로도 손쉽게 확장이 가능하다.

페르미 이전 아키텍처에서는 자주 참조되는 전역 메모리 접근시 성능을 향상시키고자 공유 메모리를 활용

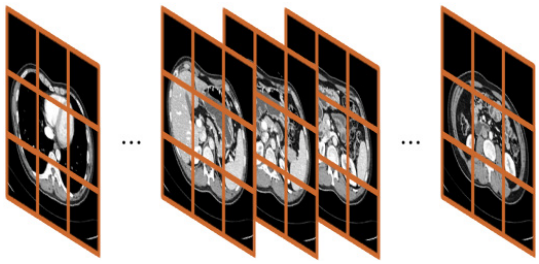


그림 3. 2차원 데이터에서 타일 구조의 예
Fig. 3. Example of tile structure for 2-D

하였다. 전역 메모리에 한번 접근한 뒤 그 데이터를 공유 메모리에 저장하여 다음 접근 시에는 공유 메모리에 접근함으로써 빠르게 접근 할 수 있었다. 또한 공유 메모리는 메모리 재사용 율을 높이기 위한 방법 이외에도 동일한 블록 공간 내에서 쓰레드 간의 통신에 사용 될 수 있다. 이 경우 쓰레드 간의 동기화가 요구 된다.

그러나 공유 메모리를 과도하게 사용하게 되면 하드웨어 제한으로 실행 할 수 있는 쓰레드 수가 감소하게 되며, 공유 메모리를 사용하기 위해서는 기존 커널 함수를 재 수정해야 하는 단점이 있다.

페르미 이후 아키텍처에서는 캐시 메모리를 지원하게 되는데 공유 메모리와 동일한 하드웨어에 위치하고 있으며 스크래치 패드(Scratch-pad Memory) 형식으로 지원하고 있다[20]. 페르미 아키텍처에서는 64 KByte의 스크래치패드 메모리를 제공한다. 이 메모리는 설정에 따라서 48 KByte공유 메모리와 16 KByte 캐시 메모리 또는 16 Kbyte 공유 메모리와 48 Kbyte 캐시 메모리로 사용할 수 있다.

캐시 메모리의 경우 CPU의 캐시 메모리와 동일한 원리로 실행된다. 전역 메모리에 접근을 시도할 때 캐시 메모리에서 검색 후 캐시 불일치(cache miss)가 발생하면 전역 메모리에서 캐시 메모리로 반입을 한다. 이러한 과정은 공유 메모리와 달리 기존 커널 함수를 수정하지 않아도 된다. 또한 현재 쓰레드에서 접근된 데이터는 다른 쓰레드에서 이웃 점에 해당하므로 캐시 적중률을 높일 수 있다. 하지만 캐시 메모리는 동일한 블록 공간 내에서 쓰레드 간의 통신에는 사용할 수 없다.

## Ⅲ. 실험환경 및 결과

### 1. 실험환경

비등방성 확산 필터의 차분방정식을 대규모 병렬 시

스템에 적용하는 두 가지 방법을 실험하였다. 첫 번째는 타일 구조 방법이고, 두 번째는 캐시 구조 방법이다. 실험에서 사용된 시스템 사양은 다음과 같다. 호스트 시스템의 CPU는 Intel Core i5-2500 3.30GHz, DDR3-12800 8GiB 이었으며, 사용된 GPU는 NVidia GeForce GTX 550 Ti 1GiB제품에서 실험하였다. 운영체제는 Windows7 Enterprise 64bit 이며 CUDA 드라이버는 5.0 버전을 사용하였다. 실험 데이터는 의료영상으로써 복부 CT 이미지를 사용하였다. 사용된 영상의 크기는 512x512x246 이었으며, 각 화소는 4바이트 크기의 단일 정밀도를 사용하였다.

비등방성 확산 필터에서 잡음 제거를 위해 기울기 함수를 사용하였다. 위 함수에서 κ값은 잡음을 정의하는 역할을 하는데 κ값에 가까울 수 록 잡음으로 판단하여 확산하게 된다. 고정된 κ의 경우 임의의 상수로 지정하여 처리하였고 이번 실험에서는 81로 지정되었다. 계산된 κ의 경우 각 반복 단계마다 전체 이미지의 평균 기울기 값을 계산하여 지정하였다. 비등방성 확산 필터를 적용하여 원하는 결과 값을 얻기 위해서는 만족할 때까지 반복하는데, 이번 실험에서는 10, 30, 50, 그리고 100회 반복으로 총 네 가지의 경우로 나누어 실험하였다.

## 2. Cyclomatic Complexity Method

공유 메모리를 사용하였을 경우 증가하는 코드 복잡도를 측정하기 위한 방법으로 사이클로매틱 복잡도(Cyclomatic Complexity) 방법을 사용하였다[24]. 이 방법에 의한 복잡도 v(G)는 그래프 형식으로 나타내며 식 (4)와 같이 표현 할 수 있다.

$$v(G) = E - N + 2 \quad (4)$$

여기서 E는 간선(Edge)의 수를 의미하고 N은 노드의 수를 의미한다. 그림 4와 그림 5는 캐시 및 공유 메모리를 이용한 “고정된 κ”의 기울기 비등방성 확산 필터 의 사코드이다. 그림 5는 Micikevicius[8]가 제안한 코드를 이용하여 작성하였고, 경계 크기는 식 (1)에서 κ에 해당한다. 그림 4와 그림 5를 사이클로매틱 복잡도의 그래프로 표현하면 그림 6과 그림 7로 표현 된다. 그림 6에서 E는 4 이고 N은 4이므로 복잡도 v(G) = 4 - 4 + 2 = 2가 된다. 또한, 그림 7의 경우, E는 12, N은 9가 되므로 복잡도 v(G) = 5가 된다. “계산된 κ”의 경우 전역 데이터의 평균값을 구하기 위해 반복문이 추가 되어 복잡도가 증가 된다. 계산된 복잡도를 표 1에 정리하였다.

```
01  function Anisotropic_cache() {
02   int3 index = {전역 메모리 위치};
03   if (index의 유효 범위 검사) {
04    전역 메모리로부터 레지스터로 데이터 반입
05    x,y,z 각 차분 값을 계산
06    전도 함수를 이용하여 변화량을 계산
07    전역 메모리으로 데이터 반출
08   }
09  }
```

그림 4 캐시 메모리를 이용한 비등방성 확산 필터 의 사코드

Fig. 4. Anisotropic diffusion filter pseudocode using cache memory.

```
01  function Anisotropic_shared() {
02   int3 index = {전역 메모리 위치 };
03   if (index의 유효 범위 검사) {
04    경계 영역의 크기를 포함하여 공유 메모리 선언
05    for( i < image.z ) {
06     if( threadIdx.y < 경계 크기 ) {
07      y축 경계 영역을 공유 메모리로 반입
08     }
09     if( threadIdx.x < 경계 크기 ) {
10      x축 경계 영역을 공유 메모리로 반입
11     }
12     전역 메모리로부터 공유 메모리로 데이터 반입
13     전도 함수를 이용하여 변화량을 계산
14     전역 메모리으로 데이터 반출
15     }
16    }
17   }
```

그림 5 공유 메모리를 이용한 비등방성 확산 필터 의 사코드

Fig. 5. Anisotropic diffusion filter pseudocode using shared memory.


그림 6. 캐시 메모리를 이용한 비등방성 확산 필터의 순서도

Fig. 6. Anisotropic diffusion filter flowchart Using cache memory.

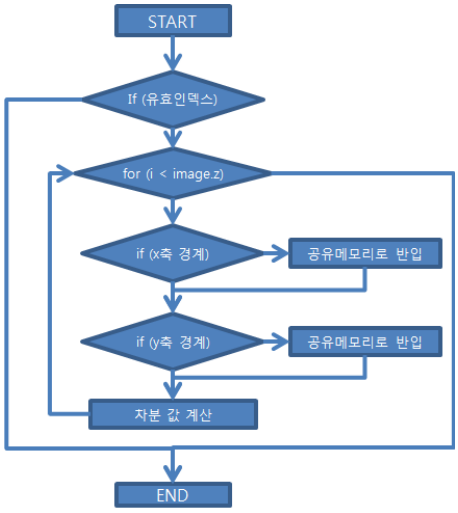



그림 7. 공유 메모리를 이용한 비등방성 확산 필터의 순서도
Fig. 7. Anisotropic diffusion filter flowchart Using shared memory

표 1. Cyclomatic Complexity 방법을 이용한 코드 복잡도 비교
Table 1. Comparison of code complexity using Cyclomatic Complexity method.

| Cyclomatic Complexity | 기울기 | |
|---|---|---|
| | 고정 κ | 계산 κ |
| 공유 메모리 | 5 | 6 |
| 캐시 메모리 | 2 | 3 |

## 3. 공유 및 캐시 메모리에서 코드 복잡도 비교

조건 분기문을 기준으로 하는 사이클로매틱 복잡도 방법은 GPU에 매우 적합하다고 볼 수 있다. GPU에서 SIMD와 유사한 방식으로 실행하는 GPU에서 조건 분기는 전체 성능을 감소시키는 경우가 발생한다. 예를 들면 CUDA에서 하나의 쓰레드는 32개를 하나의 묶음으로 작업 스케쥴링을 할당하며 이 방법을 CUDA에서는 랩(Wrap)이라 한다. 이들 중 반절이 물리적인 프로세서에 할당되어 처리 하게 된다. 만약 이 16개의 쓰레드 중 하나의 쓰레드가 조건 분기에서 참의 값을 가지고 나머지 15개의 쓰레드가 거짓의 값을 가진다면 1개의 쓰레드를 처리하기 위해 15개의 쓰레드가 대기 상태로 전환된다. 따라서 15개의 쓰레드는 작업을 멈추게 되며 수천 개의 쓰레드가 실행되는 GPU에서는 성능 감소에 영향을 미치게 된다.

표 1에 의하면 공유 메모리를 사용한 경우에 복잡도가 높게 나오는데, 이는 공유 메모리에 데이터를 반입하기 위한 조건문이 추가되었기 때문이다. 이 때 조건문은 이웃된 데이터를 공유 메모리에 반입해야하기 때문에 범위 초과(Out of Range) 구문이 추가 된 것이다.

표 1에서 모든 복잡도가 10 이하로 나타나고 있지만 위에서 설명한 것처럼 GPU에서 조건 분기가 많다는 것은 성능에 관련된 문제이다. 그러므로 캐시 메모리를 이용한 방법은 유용하다고 볼 수 있다.

## 4. 결과

표 2는 기울기 기반을 이용한 비등방성 확산 필터에서 캐시 메모리와 공유 메모리의 성능을 비교하여 보여주고 있다. GPU시스템의 병렬화 알고리즘에 상응하는 순차코드를 단일 CPU에서 실행시킨 수행시간에 대해서는 cpu열에 표기하였다.

표 2에서는 캐시 및 공유 메모리가 각각 16x16 쓰레드 구성에서 가장 우수한 성능을 보이고 있다.

그림 8은 캐시 및 공유 메모리에서 가장 우수한 성능을 보인 결과를 CPU와 성능을 비교한 것이다. 실험의 결과에서 볼 수 있듯이 메모리 병합 기술을 기반으로 캐시 메모리를 활용하였을 경우, 쓰레드 구성에 따른 성능에 차이는 있었지만 가장 우수한 성능을 보인 쓰레드 구성끼리의 차이는 두 메모리 경우 모두 유사한 성능을 보이고 있다. 그림 8에서 “고정된 κ”의 경우 캐시 메모리가 조금 더 우수한 성능을 보이고 있는데 이것은 코드 복잡도 비교에서 언급한 바와 같이 조건문으로 인한 성능 차이가 발생하였기 때문이다. 그림 8에서 “계산된 κ”의 경우 전역 메모리 접근 빈도가 고정된 κ에 비해 높아 쓰레드 실행속도가 비슷한 것으로 나타나고 있다. 즉, 쓰레드의 실행속도가 메모리 접근 지연으로 감추어졌다.

기존 CUDA 시스템에서 공유 메모리는 두 가지 목적으로 사용되어 졌다. 첫째 전역 메모리 접근 지연에 따른 실행 속도의 손실을 보안하기 위한 목적과, 둘째 동일 블록 공간에서 쓰레드 간 통신을 위한 목적으로 사용되어왔다. 하지만 이번 결과는 페르미 아키텍처 시스템에서 공유 메모리를 사용하여 전역 메모리 접근 지연을 보안하고자 할 경우 큰 이점이 없다는 결과를 얻을 수 있었다. 따라서 공유 메모리는 동일 블록 공간에서 쓰레드 간 통신을 위해서 사용하고 전역 메모리 접근 지연의 해결책은 캐시 메모리를 사용하여 성능을 보장받을 수 있음을 보여주고 있다.

표 2. 기울기 비등방성 확산 필터의 실행시간 (단위: 초)

Table 2. Execution time of gradient anisotropic diffusion filter (Unit: second)

| 반복 횟수 | 쓰레드 구성수 | 고정된 κ | | 계산된 κ | |
|---|---|---|---|---|---|
| | | 캐시 | 공유 | 캐시 | 공유 |
| 10 | 8x8 | 2.00 | 1.80 | 3.09 | 2.91 |
| | 16x16 | 1.52 | 1.56 | 2.61 | 2.67 |
| | 32x32 | 1.88 | 1.72 | 2.98 | 2.84 |
| | cpu | | 122.83 | | 137.71 |
| 30 | 8x8 | 5.49 | 5.00 | 8.35 | 8.34 |
| | 16x16 | 4.10 | 4.28 | 7.48 | 7.63 |
| | 32x32 | 5.17 | 4.76 | 8.53 | 8.14 |
| | cpu | | 367.39 | | 411.28 |
| 50 | 8x8 | 9.02 | 8.20 | 14.65 | 13.77 |
| | 16x16 | 6.71 | 7.02 | 12.34 | 12.58 |
| | 32x32 | 8.44 | 7.81 | 14.07 | 13.44 |
| | cpu | | 585.68 | | 657.06 |
| 100 | 8x8 | 17.83 | 16.35 | 29.09 | 27.49 |
| | 16x16 | 13.23 | 13.83 | 24.49 | 24.98 |
| | 32x32 | 16.68 | 15.41 | 27.94 | 26.68 |
| | cpu | | 1169.78 | | 1314.27 |

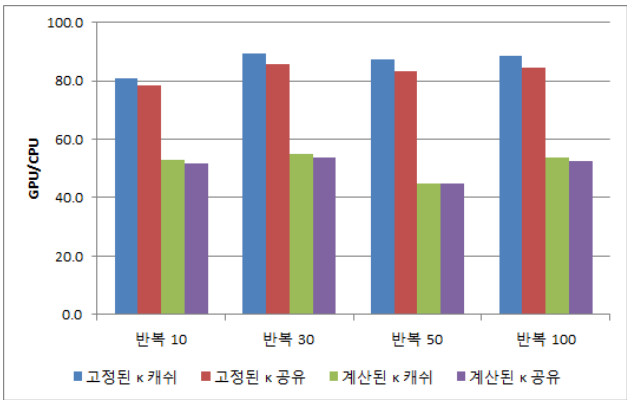



그림 8. 기울기 비등방성 확산필터에서 캐시 및 공유 메모리의 CPU 대비 성능 비교

Fig. 8. Performance of cache and shared memory algorithm for a gradient anisotropic diffusion filter compared to CPU time.

## Ⅳ. 결 론

대규모 병렬 시스템에서 메모리 접근 성능을 높이는 방법에는 두 가지로 나뉠 수 있다. 첫 번째는 공유메모리를 사용하는 방법이고, 두 번째는 캐시 메모리를 사용하는 방법이다. 공유 메모리를 사용하는 방법은 캐시 메모리가 지원되기 이전에 사용된 방법이나 현재까지도 최적화 방법으로 많이 사용되고 있다. GPU 시스템에서 공유 메모리와 캐시 메모리는 같은 하드웨어로 구현되어 있다. 그러나 공유메모리로 사용할 경우 기존 커널 함수를 재 구현해야 하며 코드의 복잡도를 증가시킨다.

캐시 메모리를 사용하는 방법은 공유 메모리와 달리 기존의 커널 함수를 재구현할 필요가 없으므로 구현의 복잡도가 발생되지 않는다. 단지 블록의 쓰레드 수를 적절하게 조합하여 최적의 값을 찾는다. 본 논문에서는 캐시 메모리를 이용하여 쓰레드 블록 구성을 적절히 설정해 줌으로써 공유 메모리와 유사한 성능을 나타낼 수 있음을 보여주었다.

## 참 고 문 헌

[1] K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, and K. Yelick, “Auto-tuning the 27-point stencil for multicore,” presented at the In Proc. iWAPT2009: The Fourth International Workshop on Automatic Performance Tuning, 2009.

[2] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, “Larrabee: a many-core x86 architecture for visual computing,” presented at the ACM SIGGRAPH 2008 papers, Los Angeles, California, 2008.

[3] NVIDA. (2012). CUDA_C_Programming_Guide (v4.2 ed.). http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

[4] A. Munshi. (2012). The OpenCL Specification (v1.2 rev15 ed.). http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

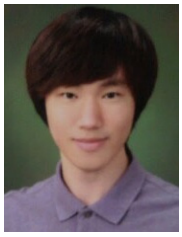
[5] D. Moth. (2011). Taming GPU compute with C++ AMP. http://channel9.msdn.com/Events/BUILD/BUILD2011/TOOL-802T

[6] M. Harris. (2002). The General-Purpose Computation on Graphics Hardware. http://www.gpgpu.org

[7] B. R. Gaster and L. Howes, “Can GPGPU Programming Be Liberated from the

Data-Parallel Bottleneck?," Computer, vol. 45, pp. 42-52, 2012.
[8] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," presented at the Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, Washington, D.C., 2009.
[9] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," presented at the Proceedings of the 2008 ACM/IEEE conference on Supercomputing, Austin, Texas, 2008.
[10] NVIDIA. (2009). FERMI Compute Architecture White Paper (v1.1 ed.). http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
[11] A. P. Witkin, "Scale-space filtering," presented at the Proceedings of the Eighth international joint conference on Artificial intelligence - Volume 2, Karlsruhe, West Germany, 1983.
[12] G. A. McMechan, "MIGRATION BY EXTRAPOLATION OF TIME-DEPENDENT BOUNDARY VALUES*," Geophysical Prospecting, vol. 31, pp. 413-420, 1983.
[13] P. Perona and J. Malik, "Scale-space and edge detection using anisotropic diffusion," Pattern Analysis and Machine Intelligence, IEEE Transactions on, vol. 12, pp. 629-639, 1990.
[14] M. J. Black, G. Sapiro, D. H. Marimont, and D. Heeger, "Robust anisotropic diffusion," Image Processing, IEEE Transactions on, vol. 7, pp. 421-432, 1998.
[15] Y. Xiaosheng, W. Chengdong, J. Tong, and C. Shuo, "A time-dependent anisotropic diffusion image smoothing method," in Intelligent Control and Information Processing (ICICIP), 2011 2nd International Conference on, 2011, pp. 859-862.
[16] A. Yezzi, Jr., "Modified curvature motion for image smoothing and enhancement," Image Processing, IEEE Transactions on, vol. 7, pp. 345-352, 1998.
[17] R. T. Whitaker and X. Xinwei, "Variable-conductance, level-set curvature for image denoising," in Image Processing, 2001. Proceedings. 2001 International Conference on, 2001, pp. 142-145 vol.3.
[18] G. Gerig, O. Kubler, R. Kikinis, and F. A. Jolesz, "Nonlinear anisotropic filtering of MRI data," Medical Imaging, IEEE Transactions on, vol. 11, pp. 221-232, 1992.
[19] NVIDIA. (2012). CUDA C BEST PRACTICES GUIDE (v4.1 ed.). http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf
[20] M. Moazeni, A. Bui, and M. Sarrafzadeh, "A memory optimization technique for software-managed scratchpad memory in GPUs," in Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on, pp. 43-49, 2009.
[21] 강동수, 신병석. "의료영상에서의 GPGPU활용.", 전자공학회지, 36권 5호, pp 79-87. 2009년 5월.
[22] 이호영, 박종현, 김준성. "CUDA를 이용한 FDTD 알고리즘의 병렬처리.", 전자공학회논문지-CI편, 47권 4호, pp 82-87. 2010년. 7월.
[23] Sung-In Choi, Soon-Yong Park, Jun Kim and Yong-Woon Park. "Multi-view Range Image Registration using CUDA." In: : 대한전자공학회, pp 733-736. 2008년 7월.
[24] McCabe, T. J. "A Complexity Measure." Software Engineering, IEEE Transactions on SE-2(4): 308-320. 1976.

## 저 자 소 개



**김 현 규**(학생회원)
2012년 한국교육개발원
컴퓨터과학과 학사 졸업.
2012년～현재 전북대학교
컴퓨터공학과 석사 과정.

<주관심분야 : 병렬처리, 영상처리>

**이 효 종**(평생회원)-교신저자
1986년 미국 유타대학교
컴퓨터과학과 학사 졸업.
1988년 미국 유타대학교
컴퓨터과학과 석사 졸업.
1991년 미국 유타대학교
컴퓨터과학과 박사 졸업.

<주관심분야 : 영상처리, 컴퓨터비젼, 의료영상, 병렬처리>