REPORT

Cache Locality Report

과목명	시스템프로그래밍	담당교수	유시환 교수님
학 번	32240633	전 공	모바일시스템공학과
이 름	김민준	제 출 일	25.10.04

Cache Locality Report - 목차

1. Introduction

1.1 연구 배경 및 목적

현대 컴퓨터 시스템에서 프로세서와 메모리 간의 속도 차이는 지속적으로 증가하고 있다. 프로세서의 연산 속도는 무어의 법칙에 따라 급격히 향상되어 왔으나 메인 메모리의 접근 속도는 상대적으로 느린 개선율을 보였다.

이러한 성능 격차를 메모리 장벽이라 하며 이를 해소하기 위해 계층적 메모리 구조가 도입되었다. 그 중심에는 프로세서와 메인 메모리 사이에 위치한 캐시 메모리가 있다. 캐시 메모리의 효율성은 프로그램의 메모리 접근 패턴이 지역성의 원리(Principle of Locality)를 얼마나 잘 따르는 가에 의해 결정된다.

지역성의 원리는 프로그램이 특정 시간대에 메모리의 제한된 영역을 집중적으로 접근하는 경향을 설명하는 개념으로 크게 시간 지역성(Temporal Locality)과 공간 지역성(Spatial Locality)으로 구분된다.

본 연구는 3 차원 배열의 순회 순서가 캐시 성능에 미치는 영향을 실증적으로 분석하고 공간 지역성과 시간 지역성의 개념을 실험적으로 검증하는 것을 목적으로 한다. 특히 동일한 연산을 수행하지만 메모리 접근 순서만 다른 두 가지 알고리즘의 성능 차이를 측정함으로써 메모리 접근 패턴 최적화의 중요성을 규명하고자 한다. 이를 통해 효율적인 소프트웨어 개발을 위한 실용적 지침을 제시하고 이론적 개념과 실제 시스템 성능 간의 관계를 명확히 하고자 한다.

2. 이론적 배경

2.1 지역성의 원리

지역성의 원리는 프로그램이 실행되는 동안 메모리 접근이 시간적, 공간적으로 집중되는 경향을 설명하는 개념이다. 이는 캐시 메모리가 효과적으로 동작할 수 있는 근본적인 이유를 제공한다. 시간 지역성은 최근에 참조된 메모리 위치가 가까운 미래에 다시 참조될 가능성이 높다는 원리이다. 이는 다음과 같은 프로그래밍 패턴에서 관찰된다. 첫째 반복문에서 루프 변수는 각 반복마다 읽고 갱신되므로 높은 시간 지역성을 갖는다. 둘째 자주 호출되는 함수의 코드와 지역 변수는 반복적으로 접근된다. 셋째 스택 프레임의 활성 함수 데이터는 함수 실행 중 지속적으로 참조된다. 시간 지역성이 높은 프로그램에서는 한 번 캐시에 로드된 데이터가 캐시에서 제거되기 전에 여러 번 재사용되므로 캐시 적중률이 증가하여 전체 성능이 향상된다.

공간 지역성은 한 메모리 위치가 참조되면 그 근처의 메모리 위치들도 곧 참조될 가능성이 높다는 원리이다. 이는 다음과 같은 상황에서 나타난다. 첫째 배열의 순차 접근에서 연속된 요소들이 차례로 접근된다. 둘째 순차적 코드 실행에서 명령어들이 메모리상 연속된 위치에 저장되어 있다. 셋째 구조체의 멤버 접근 시 인접한 필드들이 함께 사용되는 경향이 있다. 공간 지역성을 활용하면 캐시 라인 단위로 데이터를 미리 로드하여 메모리 접근 효율성을 극대화할 수 있다.

현대 컴퓨터 시스템의 캐시는 이러한 지역성의 원리를 하드웨어 레벨에서 구현한 것이다. 시간 지역성은 최근 사용된 데이터를 캐시에 보관함으로써 공간 지역성은 요청된 데이터뿐만 아니라 인접한 데이터를 함께 로드함으로써 활용된다. 따라서 프로그램의 메모리 접근 패턴이 이러한 지역성을 잘 따를수록 캐시의 효율이 높아지고 결과적으로 프로그램의 실행 속도가 향상된다.

2.2 캐시 메모리 구조

캐시 메모리는 프로세서와 메인 메모리 사이에 위치하여 빈번히 접근되는 데이터를 임시로 저장하는 고속 메모리이다. 캐시의 핵심 구조와 동작 원리를 이해하는 것은 효율적인 프로그램 작성에 필수적이다.

캐시 라인은 캐시가 데이터를 관리하는 기본 단위이다. 현대 프로세서의 캐시 라인 크기는 일반적으로 64 바이트이다. 메모리에서 데이터를 읽을 때 프로세서는 요청된 단일 바이트나 워드만을 가져오는 것이 아니라 해당 데이터를 포함하는 전체 캐시 라인(64 바이트)을 한 번에 로드한다. 이는 공간 지역성을 하드웨어 레벨에서 구현한 것으로 인접한 데이터가 곧 필요할 것이라는 가정에 기반한다. 캐시 라인의 의의는 다음과 같다. 첫째 공간 지역성을 자동으로 활용하여 한 번의 메모리 접근으로 여러 데이터를 미리 가져온다. 둘째 메모리 대역폭을 효율적으로 활용하여 개별 바이트 단위 전송보다 블록 단위 전송이 더 효율적이다. 셋째 캐시 미스의 비용을 상각하여 한 번의 미스 비용으로 여러 데이터를 획득할 수 있다.

예를 들어 64 바이트 캐시 라인은 16 개의 32 비트 정수를 담을 수 있으므로 첫 번째 정수 접근 시 캐시 미스가 발생하더라도 이후 15 개의 연속된 정수는 이미 캐시에 로드되어 빠르게 접근할 수 있다. 캐시 계층 구조는 현대 프로세서가 일반적으로 채택하는 다단계 캐시 시스템이다.

일반적인 3 단계 구조는 다음과 같다. L1 캐시는 32-64KB 크기로 프로세서 코어에 가장 가깝게 위치하며 접근 시간이 1-2 사이클로 가장 빠르다. L2 캐시는 256KB-1MB 크기로 중간 속도를 제공하며 접근 시간은 10-20 사이클이다. L3 캐시는 수 MB 크기로 여러 코어가 공유하며 접근 시간은 40-75 사이클이다. 마지막으로 메인 메모리는 GB 단위의 용량을 가지지만 접근 시간이 100-300 사이클로 매우 느리다. 이러한 계층 구조에서 캐시 미스가 발생하면 하위 계층으로 순차적으로 내려가며 데이터를 찾게 된다. L1 캐시에서 미스가 발생하면 L2 캐시를 확인하고 L2 에서도 없으면 L3 를 거쳐 최종적으로 메인 메모리까지 접근한다. 따라서 상위 캐시(특히 L1)의 적중률을 높이는 것이 전체 시스템 성능에 결정적인 영향을 미친다. 캐시 미스로 인한 메인 메모리 접근은 L1 캐시 적중에 비해 100 배 이상 느리므로 효율적인 메모리 접근 패턴 설계가 매우 중요하다.

2.3 C 언어의 배열 메모리 배치

C 언어는 다차원 배열을 메모리에 배치할 때 행 우선 순서를 사용한다. 이는 배열의 가장 오른쪽 인덱스가 메모리 주소상에서 가장 빠르게 변화하는 방식이다. 이러한 메모리 배치 방식을 이해하는 것은 캐시 효율적인 프로그램을 작성하는 데 필수적이다. 3 차원 배열 int data[I][J][K]의 경우 메모리는 다음과 같이 배치된다.

```
메모리 주소 증가 방향 →
data[0][0][0], data[0][0][1], data[0][0][2], ..., data[0][0][K-1],
data[0][1][0], data[0][1][1], data[0][1][2], ..., data[0][1][K-1],
data[0][2][0], data[0][2][1], data[0][2][2], ..., data[0][2][K-1],
...
data[0][J-1][0], ..., data[0][J-1][K-1],
data[1][0][0], data[1][0][1], ..., data[1][0][K-1],
...
data[I-1][J-1][K-1]
```

이러한 배치에서 인접한 메모리 위치는 k 인덱스가 1 씩 증가하는 요소들이다. 예를 들어, data[0][0][0]의 바로 다음 메모리 위치는 data[0][0][1]이고 그 다음은 data[0][0][2]이다. 반면 data[0][0][K-1] 다음의 메모리 위치는 data[0][1][0]이 된다.

이러한 배치 방식은 공간 지역성 활용에 중요한 함의를 갖는다. 가장 오른쪽 인덱스(k)를 순차적으로 증가시키는 접근 패턴은 물리적으로 연속된 메모리 위치를 순회하게 되어 최적의 공간 지역성을 제공한다. 반대로 왼쪽 인덱스(i 또는 j)를 먼저 증가시키는 패턴은 메모리상에서 멀리 떨어진 위치를 접근하게 되어 공간 지역성이 저하된다.

구체적으로 data[i][j][k]에서 data[i][j][k+1]로 이동하면 메모리 주소가 4 바이트(int 크기)만큼 증가한다. 그러나 data[i][j][k]에서 data[i][j+1][k]로 이동하면 K×4 바이트만큼 점프하게 된다. 본 실험에서 K=16 이므로 j 를 증가시키는 것은 64 바이트(16×4)씩 떨어진 위치로 이동하는 것을 의미한다. 이는 정확히 하나의 캐시 라인 크기에 해당하므로 각 접근마다 새로운 캐시 라인이 필요하게 되어 캐시 효율이 급격히 저하된다.

따라서 C 언어에서 다차원 배열을 효율적으로 순회하려면 가장 안쪽 루프에서 가장 오른쪽 인덱스를 증가시켜야 한다. 이는 단순한 코딩 스타일의 문제가 아니라 하드웨어 아키텍처와 메모리 배치 방식에 기반한 성능 최적화의 핵심 원칙이다.

3. 실험 설계 및 방법

3.1 환경실험 및 알고리즘

본 실험은 메모리 접근 패턴이 캐시 성능에 미치는 영향을 정량적으로 측정하기 위해 설계되었다. 실험 환경은 다음과 같이 구성되었다.

```
// Array size definitions
#define I_MAX 0x1000  // 4096
#define J_MAX 0x1000  // 4096
#define K_MAX 0x10    // 16
```

데이터 구조: 3 차원 정수형 배열 int data[0x1000][0x1000][0x10]을 사용하였다. 이는 각 차원의 크기가 4096×4096×16 이며 총 268,435,456 개의 요소를 포함한다. 각 요소는 32 비트 정수형(4 바이트)이므로 전체 메모리 사용량은 1,073,741,824 바이트 즉 약 1GB 이다. 이러한 대용량 배열을 선택한 이유는 캐시 크기를 충분히 초과하여 캐시 효과를 명확히 관찰하기 위함이다.

```
// Array initialization
printf("Initializing array with 1...\n");
for (int i = 0; i < I_MAX; i++) {
    for (int j = 0; j < J_MAX; j++) {
        for (int k = 0; k < K_MAX; k++) {
            data[i][j][k] = 1;
        }
    }
}
printf("Array initialization complete.\n\n");
```

실험 환경: Ubuntu Linux 운영체제에서 실험을 수행하였으며 gcc 컴파일러를 사용하여 코드를 컴파일하였다. 시간 측정은 gettimeofday() 함수를 이용하여 마이크로초 단위의 정밀도로 수행하였다. 모든 배열 요소는 실험 전에 1 로 초기화하여 일관된 조건을 유지하였다.

실험 알고리즘: 동일한 연산(배열의 모든 요소 합산)을 수행하되 순회 순서만 다른 두 가지 버전을 구현하였다.

Version 1 은 (i, j, k) 순서로 순회한다:

```
// Version 1 (i,j,k)
printf("Version 1 (i,j,k)\n");
sum = 0;

gettimeofday(&start_tv, NULL);

for (int i = 0; i < I_MAX; i++) {
    for (int j = 0; j < J_MAX; j++) {
        for (int k = 0; k < K_MAX; k++) {
            sum += data[i][j][k];
        }
    }
}

gettimeofday(&end_tv, NULL);
```

이 버전은 가장 안쪽 루프에서 k 인덱스를 증가시킨다. C 언어의 행 우선 배치 방식에서 k 는 가장 오른쪽 인덱스이므로 이는 메모리상 연속된 위치를 순차적으로 접근하는 패턴이다. 따라서 최적의 공간 지역성을 제공한다.

Version 2 는 (k, i, j) 순서로 순회한다:

```
// Version 2 (k,i,j)
printf("Version 2 (k,i,j)\n");
sum = 0;

gettimeofday(&start_tv, NULL);

for (int k = 0; k < K_MAX; k++) {
    for (int i = 0; i < I_MAX; i++) {
        for (int j = 0; j < J_MAX; j++) {
            sum += data[i][j][k];
        }
    }
}

gettimeofday(&end_tv, NULL);
```

이 버전은 가장 안쪽 루프에서 j 인덱스를 증가시킨다. 이는 메모리상에서 불연속적인 위치를 접근하게 되어 공간 지역성이 저하된다. 구체적으로, data[i][j][k]에서 data[i][j+1][k]로 이동할 때 메모리 주소가 K×4 = 64 바이트씩 점프하게 되어, 각 접근마다 새로운 캐시 라인이 필요하게 된다.

두 버전 모두 총 268,435,456 번의 메모리 접근과 덧셈 연산을 수행하며 최종 결과는 동일하게 268,435,456 이 되어야 한다. 이는 실험의 정확성을 검증하는 수단으로 사용된다.

3.2 측정 방법론

정확한 성능 측정을 위해 다음과 같은 체계적인 방법론을 적용하였다.

초기화 단계: 실험 시작 전에 모든 배열 요소를 1 로 초기화한다. 이는 3 중 루프를 사용하여 수행되며 초기화 시간은 측정 대상에서 제외된다.

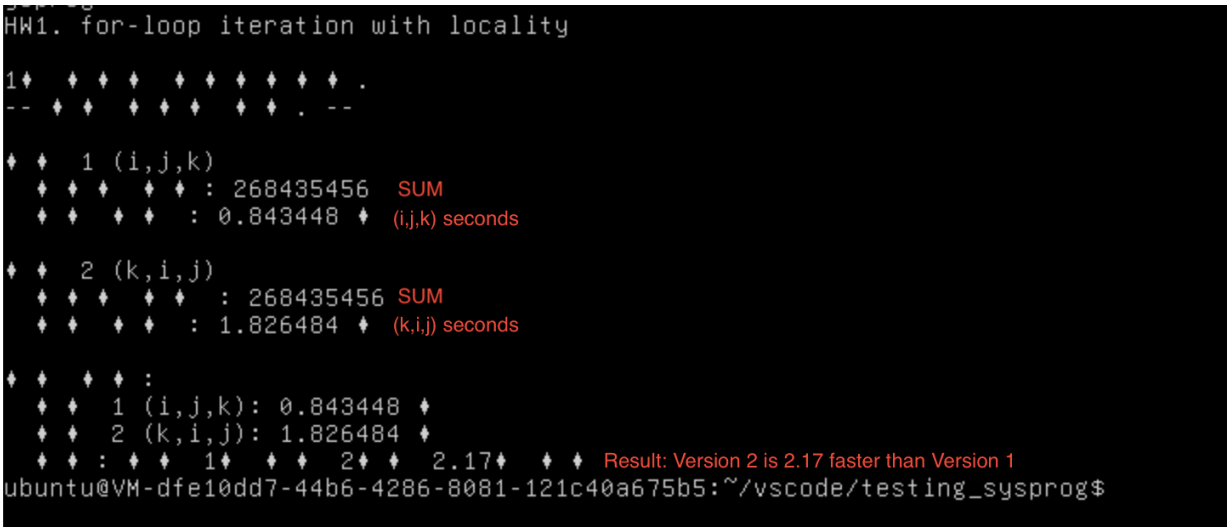
시간 측정: 각 버전의 실행 시간 측정은 다음 절차를 따른다. gettimeofday() 함수를 호출하여 시작 시간을 기록한 후 전체 배열을 순회하며 합산 연산을 수행한다. 이 함수는 마이크로초(10^-6 초) 단위의 정밀도를 제공하여 정확한 측정이 가능하다. 연산 완료 후 다시 gettimeofday()를 호출하여 종료 시간을 기록하고 두 시간의 차이를 계산하여 경과 시간을 구한다. 시간 차이 계산 시 마이크로초 부분이 음수가 되는 경우 1,000,000 마이크로초를 더하고 초 부분에서 1 을 빼는 처리를 수행하여 정확한 결과를 얻는다.

결과 검증: 계산된 합계가 예상값(268,435,456)과 일치하는지 확인하여 알고리즘이 올바르게 동작했음을 검증한다. 또한 두 버전의 합계가 동일한지 확인하여 순회 순서와 무관하게 동일한 연산을 수행했음을 보장한다.

성능 비교: 두 버전의 실행 시간을 비교하여 성능 비율을 계산한다. 이를 통해 메모리 접근 패턴의 차이가 실제 성능에 미치는 영향을 정량화한다.

이러한 측정 방법론은 외부 요인의 영향을 최소화하고, 순수하게 메모리 접근 패턴에 따른 캐시 성능 차이를 측정할 수 있도록 설계되었다.

4. 실험 결과



4.1 정량적 결과

실험을 통해 얻은 정량적 결과는 다음 표와 같다.

Version	순회 순서	실행 시간 (초)	합계	기준 대비 성능
Version 1	(i,j,k)	0.843448	268,435,456	1.00x (기준)
Version 2	(k,i,j)	1.826484	268,435,456	2.17x (느림)

두 버전 모두 정확히 268,435,456 개의 요소를 순회하였으며, 합계 결과도 동일하게 268,435,456 으로 일치하여 알고리즘의 정확성이 검증되었다. 그러나 실행 시간에는 현저한 차이가 나타났다.

Version 1 은 0.843448 초가 소요되었으며 이를 기준으로 설정하였다. Version 2 는 1.826484 초가 소요되어 Version 1 보다 약 0.983 초 더 오래 걸렸다. 이는 상대적으로 약 116.5% 증가한 수치이다. 다시 말해 Version 2 는 Version 1 에 비해 약 **2.17 배 느린** 성능을 보였다.

이러한 결과는 메모리 접근 패턴의 차이만으로도 2 배 이상의 성능 차이가 발생할 수 있음을 명확히 보여준다. 두 버전은 동일한 수의 연산을 수행하고 동일한 데이터에 접근하며 동일한 결과를 산출하지만 단지 메모리를 순회하는 순서만 다를 뿐이다. 이는 알고리즘의 시간 복잡도가 O(n)으로 동일하더라도, 실제 실행 시간은 메모리 접근 패턴에 크게 영향을 받는다는 것을 의미한다.

4.2 성능 차이 분석

실행 시간의 절대적 차이는 약 0.983 초이다. 이를 총 메모리 접근 횟수 268,435,456 번으로 나누면 메모리 접근 1 회당 평균 약 3.66 나노초의 추가 지연이 발생한 것으로 계산된다.

현대 프로세서의 클럭 속도가 일반적으로 2-4 GHz 범위임을 고려하면 3.66 나노초는 대략 7-15 사이클에 해당한다. 이러한 성능 차이의 주요 원인은 캐시 미스율의 차이이다. Version 1 은 93.75%의 높은 캐시 적중률을 달성하여 16 번의 메모리 접근 중 15 번이 L1 캐시에서 즉시 처리되었다. 반면 Version 2 는 L1 캐시에서 거의 모든 접근이 캐시 미스를 발생시켰을 것으로 추정된다. 각 접근마다 64 바이트씩 떨어진 메모리 위치로 점프하는 패턴으로 인해 이전에 로드된 캐시 라인을 재활용할 수 없었기 때문이다.

측정된 접근당 7-15 사이클의 추가 지연은 L1 캐시 적중(1-2 사이클)과 메인 메모리 접근(100-300 사이클)의 중간 수준으로 이는 Version 2 의 캐시 미스 상당수가 L2 캐시(10-20 사이클) 또는 L3 캐시(40-75 사이클)에서 처리되었음을 시사한다. L1 캐시에서는 미스가 발생했지만 규칙적인 스트라이드 패턴을 하드웨어 프리페처가 감지하여 필요한 데이터를 미리 L2/L3 캐시로 가져왔을 가능성이 높다.

5. 토의 및 분석

5.1 공간지역성 관점의 분석

공간 지역성은 한 메모리 위치가 참조되면 그 근처의 메모리 위치들도 곧 참조될 가능성이 높다는 원리이다. 본 실험의 두 버전은 공간 지역성 측면에서 극명한 차이를 보인다.

Version 1 의 공간 지역성 분석

Version 1 은 가장 안쪽 루프에서 k 인덱스를 0 부터 15 까지 순차적으로 증가시킨다. C 언어의 행 우선 배치에서 k 는 가장 오른쪽 인덱스이므로, 이는 메모리상 물리적으로 연속된 위치를 순서대로 접근하는 것을 의미한다.

구체적인 메모리 접근 패턴은 다음과 같다:

메모리 주소: 0x0000 → 0x0004 → 0x0008 → 0x000C → ... → 0x003C

배열 요소: [0][0][0] → [0][0][1] → [0][0][2] → ... → [0][0][15]

각 정수형 요소는 4 바이트를 차지하므로 k 가 0 부터 15 까지 증가하면 총 64 바이트의 연속된 메모리 영역을 순회한다. 이는 정확히 현대 프로세서의 표준 캐시 라인 크기에 해당한다.

캐시 라인 활용 관점에서 Version 1 의 동작을 살펴보면 다음과 같다. 첫 번째 접근 data[0][0][0]이 발생하면 캐시 미스가 일어나고 메모리 시스템은 해당 주소를 포함하는 64 바이트 캐시 라인 전체를 캐시로 로드한다. 이 캐시라인에는 data[0][0][0]부터 data[0][0][15]까지 16 개의 정수가 포함된다. 이후 k 가 1 부터 15 까지 증가하며 접근하는 15 번의 메모리 참조는 모두 이미 캐시에 로드된 데이터를 사용하므로 캐시 적중(Cache Hit)이 발생한다.

따라서 Version 1 의 캐시 적중률은 15/16 = 93.75%에 달한다. 16 번의 연속된 접근 중 첫 번째만 캐시 미스가 발생하고, 나머지 15 번은 모두 캐시에서 빠르게 데이터를 읽어올 수 있다. 이러한 패턴은 j 와 i 루프를 따라 전체 배열 순회에 걸쳐 일관되게 반복된다.

메모리 대역폭 관점에서도 Version 1 은 최적의 효율을 보인다. 메모리에서 64 바이트를 로드하면 그 중 64 바이트 전체를 실제로 사용하므로 메모리 대역폭 활용률은 100%이다. 이는 메모리 버스의 부하를 최소화하고, 다른 프로세스나 스레드가 메모리 시스템을 사용할 수 있는 여유를 제공한다.

Version 2 의 공간 지역성 분석

Version 2 는 가장 안쪽 루프에서 j 인덱스를 증가시킨다. 이는 메모리상에서 불연속적인 위치를 접근하는 패턴을 만든다.

구체적인 메모리 접근 패턴은 다음과 같다:

메모리 주소: 0x0000 → 0x0040 → 0x0080 → 0x00C0 → ...

배열 요소: [0][0][0] → [0][1][0] → [0][2][0] → [0][3][0] → ...

점프 크기: 64 바이트 64 바이트 64 바이트

data[i][j][k]에서 data[i][j+1][k]로 이동할 때 메모리 주소는 K × 4 바이트 = 16 × 4 = 64 바이트씩 증가한다. 이는 정확히 하나의 캐시 라인 크기만큼 점프하는 것을 의미한다.

캐시 라인 활용 관점에서 Version 2 의 동작을 분석하면 다음과 같다. data[i][j][k] 접근 시 캐시 미스가 발생하여 64 바이트 캐시 라인이 로드된다. 이 캐시 라인에는 data[i][j][0]부터 data[i][j][15]까지 포함되어 있다. 그러나 현재 접근은 data[i][j][k] 하나만 사용하고 다음 접근은 data[i][j+1][k]로 이동한다. 이는 완전히 다른 캐시 라인에 속하므로 다시 캐시 미스가 발생한다. 로드된 캐시 라인의 나머지 15 개 요소(data[i][j][0]부터 data[i][j][k-1], data[i][j][k+1]부터 data[i][j][15])는 전혀 사용되지 않고 버려진다.

결과적으로 Version 2 는 거의 모든 메모리 접근에서 캐시 미스가 발생한다. 이론적 캐시 적중률은 거의 0%에 가깝다. 실제로는 하드웨어 프리페칭이나 상위 레벨 캐시(L2, L3)에서 일부 적중이 발생할 수 있지만 L1 캐시 수준에서는 극도로 비효율적인 패턴이다.

메모리 대역폭 관점에서 Version 2 는 심각한 낭비를 초래한다. 64 바이트를 로드하지만 그 중 4 바이트만 사용하므로 메모리 대역폭 활용률은 6.25%에 불과하다. 이는 메모리 버스에 불필요한 트래픽을 발생시켜 시스템 전체의 메모리 성능을 저하시킬 수 있다.

두 버전의 비교 요약

항목	Version 1 (i,j,k)	Version 2 (k,i,j)
메모리 접근 패턴	연속적 (4바이트씩)	불연속적 (64바이트씩 점프)
캐시 적중률	93.75%	~0%
캐시 라인 활용	16개 중 16개 사용	16개 중 1개만 사용
메모리 대역폭 효율	100%	6.25%
공간 지역성	최적	최악

이러한 분석은 공간 지역성이 캐시 성능에 미치는 결정적 영향을 명확히 보여준다. 동일한 데이터에 접근하더라도 접근 순서에 따라 캐시 효율이 극적으로 달라지며, 이는 실행 시간에 직접적으로 반영된다.

5.2 시간 지역성 관점의 분석

시간 지역성은 최근에 참조된 메모리 위치가 가까운 미래에 다시 참조될 가능성이 높다는 원리이다. 본 실험에서 시간 지역성의 역할을 분석하면 다음과 같다.

배열 요소의 시간 지역성

두 버전 모두 각 배열 요소 data[i][j][k]를 정확히 한 번만 접근한다. 한 번 읽어서 합계에 더한 후 다시는 그 요소를 참조하지 않는다. 따라서 배열 데이터 자체는 시간 지역성이 없다**.** 이는 스트리밍(streaming) 접근 패턴으로 데이터를 한 번만 읽고 지나가는 형태이다.

이러한 상황에서는 시간 지역성보다 공간 지역성이 훨씬 중요한 역할을 한다. 각 요소를 재사용하지 않으므로 캐시에 오래 보관할 필요가 없다. 대신 한 번 로드할 때 인접한 데이터를 함께 가져와 즉시 사용하는 것이 효율적이다.

루프 변수의 시간 지역성

루프 변수 i, j, k 는 높은 시간 지역성을 보인다. 각 변수는 루프의 매 반복마다 읽히고 갱신된다. 예를 들어 Version 1 에서 변수 k 는 내부 루프에서 16 번 연속으로 읽히고 수정된다. 변수 j 는 4096 번 변수 i 는 4096×4096 번 반복적으로 접근된다.

이러한 루프 변수들은 일반적으로 프로세서의 레지스터에 할당되거나 최소한 L1 캐시에 항상 상주한다. 따라서 루프 변수의 접근은 거의 지연 없이 수행되며 두 버전 모두에서 동일한 수준의 효율을 보인다.

누적 변수의 시간 지역성

누적 변수 sum 은 모든 반복에서 읽고 쓰이므로 극도로 높은 시간 지역성을 갖는다. 268,435,456 번의 모든 반복에서 sum 을 읽어 값을 더하고 다시 sum 에 저장한다.

컴파일러는 일반적으로 이러한 누적 변수를 레지스터에 할당하여 최적화한다. 레지스터 접근은 1 사이클 내에 완료되므로 sum 변수의 접근은 거의 성능 오버헤드를 발생시키지 않는다. 이 역시 두 버전에서 동일하게 적용된다.

시간 지역성의 영향 평가

두 버전을 비교할 때 시간 지역성 측면에서는 본질적인 차이가 없다. 양쪽 모두 배열 요소는 한 번만 접근하고 루프 변수와 누적 변수는 동일한 패턴으로 반복 접근한다. 따라서 관찰된 2.17 배의 성능 차이는 시간 지역성의 차이로는 설명될 수 없다.

결론: 공간 지역성의 지배적 역할

본 실험에서 성능 차이는 주로 공간 지역성의 차이에 기인한다. 시간 지역성은 두 버전에서 유사하게 나타나므로 성능에 결정적 영향을 미치지 않는다. 이는 스트리밍 데이터 처리와 같이 데이터를 한 번만 접근하는 알고리즘에서는 메모리 접근 순서(공간 지역성)가 성능의 핵심 결정 요인임을 시사한다.

일반적으로 시간 지역성은 동일한 데이터를 반복적으로 접근하는 알고리즘(예: 행렬 곱셈, 반복 알고리즘)에서 중요한 역할을 한다. 그러나 본 실험과 같은 단일 순회(single-pass) 알고리즘에서는 공간 지역성이 훨씬 더 중요하다.

6. 결론

본 연구는 간단한 3 차원 배열 순회 실험을 통해 메모리 접근 패턴이 프로그램 성능에 얼마나 결정적인 영향을 미치는지 실증적으로 규명했다. 놀랍게도 동일한 연산을 수행하는 두 알고리즘이 단지 순회 순서의 차이만으로 2.17 배의 성능 차이를 보였다. 이는 데이터에 어떻게 접근하는지가 하드웨어의 캐시 효율성 특히 공간 지역성(spatial locality)과 직결되어 프로그램의 속도를 좌우한다는 사실을 명확히 입증한다.

이 결과는 소프트웨어 개발자에게 중요한 메시지를 전달한다. 단순히 기능 구현에만 집중하는 것을 넘어 자신이 작성한 코드가 하드웨어 위에서 어떻게 동작할지 이해하고 설계하는 것이 필수적이라는 점이다. 하드웨어 아키텍처를 고려한 설계는 그렇지 않은 설계보다 2 배 이상 빠른 성능을 이끌어낼 수 있다.

궁극적으로 뛰어난 소프트웨어는 우수한 알고리즘과 자료구조뿐만 아니라 하드웨어 아키텍처에 대한 깊이 있는 이해를 바탕으로 탄생한다. 본 실험은 이 요소들이 어떻게 상호작용하여 실제 성능을 결정하는지 보여주는 구체적인 사례로서 미래의 개발자를 교육하고 현재의 실무자들에게 영감을 주는 의미 있는 통찰을 제공한다.

```c
#include <stdio.h>
 #include <sys/time.h>

 // Array size definitions
 #define I_MAX 0x1000    // 4096
 #define J_MAX 0x1000    // 4096
 #define K_MAX 0x10       // 16

 // 3D array declaration
 int data[I_MAX][J_MAX][K_MAX];

 int main(int argc, char *argv[]) {
        struct timeval start_tv;
        struct timeval end_tv;
        struct timeval diff;
        long long sum = 0;

        printf("HW1. for-loop iteration with locality₩n₩n");

        // Array initialization
        printf("Initializing array with 1...₩n");
        for (int i = 0; i < I_MAX; i++) {
                for (int j = 0; j < J_MAX; j++) {
                        for (int k = 0; k < K_MAX; k++) {
                                data[i][j][k] = 1;
                        }
                }
        }
        printf("Array initialization complete.₩n₩n");

        // Version 1 (i,j,k)
        printf("Version 1 (i,j,k)₩n");
        sum = 0;

        gettimeofday(&start_tv, NULL);

        for (int i = 0; i < I_MAX; i++) {
                for (int j = 0; j < J_MAX; j++) {
                        for (int k = 0; k < K_MAX; k++) {
                                sum += data[i][j][k];
```

```
            }
        }
    }

    gettimeofday(&end_tv, NULL);

    // Calculate time difference
    diff.tv_sec = end_tv.tv_sec - start_tv.tv_sec;
    diff.tv_usec = (end_tv.tv_usec - start_tv.tv_usec < 0)
                        ? end_tv.tv_usec - start_tv.tv_usec + 1000000
                        : end_tv.tv_usec - start_tv.tv_usec;

    if (end_tv.tv_usec - start_tv.tv_usec < 0) {
        diff.tv_sec--;
    }

    printf("Sum: %lld₩n", sum);
    printf("Time: %ld.%06ld seconds₩n₩n", diff.tv_sec, diff.tv_usec);

    double time_v1 = diff.tv_sec + diff.tv_usec / 1000000.0;

    // Version 2 (k,i,j)
    printf("Version 2 (k,i,j)₩n");
    sum = 0;

    gettimeofday(&start_tv, NULL);

    for (int k = 0; k < K_MAX; k++) {
        for (int i = 0; i < I_MAX; i++) {
            for (int j = 0; j < J_MAX; j++) {
                sum += data[i][j][k];
            }
        }
    }

    gettimeofday(&end_tv, NULL);

    // Calculate time difference
    diff.tv_sec = end_tv.tv_sec - start_tv.tv_sec;
    diff.tv_usec = (end_tv.tv_usec - start_tv.tv_usec < 0)
```

```
                        ? end_tv.tv_usec - start_tv.tv_usec + 1000000
                        : end_tv.tv_usec - start_tv.tv_usec;

    if (end_tv.tv_usec - start_tv.tv_usec < 0) {
        diff.tv_sec--;
    }

    printf("Sum: %lld₩n", sum);
    printf("Time: %ld.%06ld seconds₩n₩n", diff.tv_sec, diff.tv_usec);

    double time_v2 = diff.tv_sec + diff.tv_usec / 1000000.0;

    // Performance Comparison
    printf("Performance Comparison:₩n");
    printf("Version 1 (i,j,k): %.6f seconds₩n", time_v1);
    printf("Version 2 (k,i,j): %.6f seconds₩n", time_v2);

    if (time_v1 < time_v2) {
        printf("Result: Version 1 is %.2fx faster than Version 2₩n", time_v2 / time_v1);
    } else if (time_v2 < time_v1) {
        printf("Result: Version 2 is %.2fx faster than Version 1₩n", time_v1 / time_v2);
    } else {
        printf("Result: Both versions have the same performance₩n");
    }

    return 0;
}
```