**멀티코어 컴퓨팅 HW #4**

2010-11904

최재민

**[1. A.]**

성능 분석을 하기 이전에 캐시 라인의 크기를 알아보기 위해 다음과 같은 명령어를 이용하였다.



**<그림 1-1. 캐시 라인 크기 측정>**

그림에서 알 수 있듯이 현재 머신의 캐시 라인 크기는 64 바이트이다. 매트릭스의 각 원소는 float 타입이므로 4바이트이고, 한 캐시 라인에 16개의 원소들이 들어갈 수 있다. 원활한 실험을 위해 매트릭스의 크기를 128 X 128로 고정하였으며 (NDIM을 128로 수정), 한 행을 위해서는 128 / 16 = 8개의 캐시 라인이 필요하다. 또한, 코드 수행시마다 걸리는 시간이 다르므로 이에 의한 오차를 최대한 줄이기 위해 1000번의 iteration을 돌린 수행 시간을 1000으로 나누어 각 매트릭스 곱셉 방법의 수행 시간을 측정하였다. 이를 위해 mat\_mul 함수 내부에 iter라는 변수를 선언하여 기존 루프들 바깥에 새로운 루프를 생성하였다. 실험 과정은 먼저 코드를 수정한 후 make 명령을 통해 컴파일하고 ./mat\_mul 을 입력하여 프로그램을 실행하였다.

**1-1. ijk & jik**

Lecture PPT에 있는 그대로 sum변수를 사용하여 ijk 방식을 구현하게 되면 각 innermost loop 당 2번의 load과 0번의 store가 일어나게 되며, 뼈대코드를 사용하면 3번의 load와 1번의 store가 일어나게 된다. 다른 방식들(kij와 jki 등)은 2번의 load와 1번의 store가 일어나므로, ijk 방식을 어떤 방법으로 구현을 하든 간에 load와 store 개수에 의한 차이는 존재할 수 밖에 없다는 것을 알아두자.

주어 코드에서 innermost loop을 분석해 보면, 행렬 a의 경우 compulsory miss rate가 (float size)/(cache line size) = 4/64 = 1/16이고, 행렬 b의 경우 매 iteration마다 miss가 나므로 compulsory miss rate는 1, 행렬 c의 경우 맨 처음 iteration에서만 miss가 나므로 compulsory miss rate는 0에 가깝다. 이들을 합치면 각 innermost loop 당 compulsory miss rate의 합은 17/16 정도가 된다.



**<그림 1-2. ijk & jik의 실행 시간>**

위 코드를 컴파일 후 실행한 결과 위와 같은 수행시간이 출력되며, 이는 1000번의 iteration을 거친 결과이므로 1000으로 나눠주면 수행시간은 평균적으로 0.00318초 정도가 된다. 현재 구현 방식으로는 다른 방식들에 비해 load가 1번 더 많으므로 약간 더 느리게 나타난 결과라고 볼 수 있다.

**1-2. kij & ikj**

이 방식을 이용하면 행렬 b의 원소가 고정된 상태에서 innermost loop이 실행되게 된다. 즉, 행렬 a의 compulsory miss rate는 0이다. 행렬 b와 c는 j가 하나씩 증가하면서 같은 행의 열들을 차례대로 읽어오므로 compulsory miss rate는 각각 1/16이 된다. 이들을 합치면 각 innermost loop 당 compulsory miss rate의 합은 2/16이 되고, 이는 Ijk 방식보다 작으므로 수행시간이 빠를 것으로 예상된다.



**<그림 1-3. kij & ikj의 실행 시간>**

실제 수행 결과는 위와 같으며, 이 역시 1000으로 나눠주면 평균 수행 시간은 0.00146초 정도이고, ijk 방식보다 약 2배 정도 빠른 것을 알 수 있다.

**1-3. jki & kji**

이 방식을 이용하면 행렬 b의 원소가 고정된 상태에서 innermost loop이 실행되므로 행렬 b의 compulsory miss rate는 0이다. 행렬 a와 c는 i가 하나씩 증가하면서 같은 열의 원소들을 읽으므로 compulsory miss rate는 각각 1이 된다. 이들을 합치만 각 innermost loop당 compulsory miss rate의 합은 2가 되며, 이는 모든 방식 중 가장 크므로 가장 느린 수행속도를 보일 것이라고 예측할 수 있다.



**<그림 1-4. jki & kji의 실행 시간>**

실제 수행 결과는 위와 같으며, 이를 1000으로 나눠주면 평균 수행 시간은 0.00691초 정도이고 이는 모든 방식 중 가장 느린 속도이다.

**[1. B.]**

Tiling은 주어진 데이터를 일정한 크기의 블락 단위로 나누어 한 그룹의 블락들이 캐시에 올라올 수 있도록 함으로써 계산 속도를 향상시기는 방법이다. lscpu 명령을 이용하면 현재 사용하는 컴퓨터의 캐시 크기를 알 수 있는데, 실험 환경에서의 캐시 크기는 다음과 같았다.



**<그림 1-5. 실험 환경에서의 캐시 크기>**

이 정보를 바탕으로 먼저 각 캐시에 한 그룹의 블락들이 올라올 수 있는 블락의 크기를 계산해보자. 행렬곱을 위해서는 행렬 a와 b에서 각각 한 블락을 읽어야 하며, 행렬 c의 한 블락도 있어야 한다. 또 블락 내 한 원소의 타입은 float이므로 그 크기는 4바이트이다. 따라서 한 번의 Tiling에 필요한 메모리 공간의 크기는 3 \* B^2 \* 4 = 12 \* B^2 바이트이며, 이 크기가 캐시 여유공간의 크기보다 작아야 Tiling이 원활히 일어날 수 있는 것이다. 실험용 프로그램이 캐시의 크기 전체를 사용 가능하다고 가정한다면 각 캐시 레벨을 활용할 수 있는 블락의 크기는 다음과 같다.

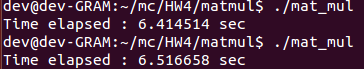
|  |  |  |
| --- | --- | --- |
| **캐시 레벨** | **캐시 크기** | **B의 범위** |
| L1d | 32KB | 12 \* B^2 < 32 \* 1024에서 B < 52.xx이므로 **B = 1 ~ 52** |
| L2 | 256KB | 12 \* B^2 < 256 \* 1024에서 B < 147.xx이므로 **B = 53 ~ 147** |
| L3 | 3072KB | 12 \* B^2 < 3072 \* 1024에서 B < 512이므로 **B = 148 ~ 511** |

**<표 1-1. 캐시 레벨과 B의 범위의 이론적 상관관계>**

다시 말하면 B가 1에는 52사이일 때는 L1d만을 활용해도 되므로 가장 속도가 빠를 것이고, B가 53에서 147사이일 때는 L2를 활용하므로 조금 속도가 느려질 것이며, 148에서 511사이일 때는 L3를 활용하므로 더 느려질 것이다. 만약 B가 512이상이 된다면 메인 메모리를 접근해야 한다. 하지만 여기까지의 계산은 완전히 이론적인 것이며, 실제로는 다른 프로그램들도 캐시를 사용하므로 실제 실험용 프로그램이 활용 가능한 캐시의 공간은 이보다 훨씬 작게 되어 B의 크기 또한 위의 표에서보다는 작아질 것이다.

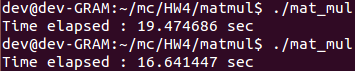
결국 이번 실험에서 할 수 있는 것은 블락 크기를 변화시켜가며 성능을 측정해보고, 수행 시간이 줄어들다가 다시 증가하는 부분을 포착하여 해당 블락 크기가 성능 변화의 threshold라는 것을 알아내는 것이다. 실험을 위해서 NDIM을 1024로 설정하였고, 블락 크기를 -s <block size> 과 같은 형식의 argument로 받기 위하여 코드를 추가한 후 블락 크기를 증가시켜가며 실험을 진행하였다.

먼저 코드를 전혀 수정하지 않은 상태에서 ijk 방식의 1024 X 1024 행렬곱을 수행한 시간은 다음과 같다.



**<그림 1-6. 기존 코드의 실행 시간>**

Tiling을 사용할 수 있도록 for loop 부분을 고쳐 만든 코드의 실행 시간은 다음과 같다. 이 경우에는 블락 크기를 1로 설정하여 기존 코드와 행렬곱의 방식은 같지만 Tiling에 필요한 overhead 때문에 월등히 느린 것을 볼 수 있다. (루프 종료 조건 확인 수의 증가와 그에 필요한 MIN 매크로 등)



**<그림 1-7. Tiling을 사용할 수 있도록 수정한 코드의 실행 시간>**

이후 블락 크기를 512까지 하나씩 증가시켜가며 ijk 방식의 1024 X 1024 행렬곱의 수행시간을 측정하였으며, 그 결과는 다음과 같은 그래프로 나타낼 수 있다. (자세한 수행시간 값들은 matmul\_result.xlsx파일에 저장되어 있다) 이론적으로도 블락 크기가 512이상일 경우 메인 메모리로의 접근이 필요하고, 실제로는 그 이하일 때 메인 메모리를 접근하게 되므로 그 이상을 실험할 필요성은 없다고 판단하였다.

**<그림 1-8. 블락 크기에 따른 실행 시간과 속도>**

실험 방법은 shell script를 이용하여 ./mat\_mul -s <block size> >> result.txt를 block size를 하나씩 증가시켜가며 실행하도록 하였고, 실행이 끝난 후 result.txt에 있는 시간 값들을 엑셀 파일에 복사하여 그래프를 제작하였다.

Tiling을 사용할 때의 캐시 미스의 횟수는이므로, 같은 캐시 레벨을 사용하는 B의 범위 내에서는 이 공식에 따라 성능이 결정된다. m은 한 캐시 라인에 들어가는 원소의 개수므로 64B / 4B = 16으로 일정하고, N 역시 1024로 일정하므로 실질적으로 캐시 미스의 횟수는 B에 비례하는게 되어 B가 증가할수록 캐시 미스의 횟수 역시 증가하게 된다. 그래프에서 B가 증가할수록 조금씩 속도가 감소하는 것이 이 때문이다. 또 그래프에서 B가 8~10 근처일 때까지 속도가 급격히 증가하는 것을 볼 수 있는데, Tiling의 overhead로 인한 성능 저하 효과가 줄어드는 것이 원인으로 생각된다. 그 이후에는 속도가 점점 감소하는데, 이는 더 낮은 레벨의 캐시를 사용해야 한다는 것을 의미한다. 그 이후에는 매우 미약하게 속도가 감소하다가 B가 240 근처에서 성능이 소폭 저하되고 fluctuation이 이전보다 크게 일어나기 시작하는데, 이 역시 낮은 레벨의 캐시를 사용하는 데서 오는 효과라고 생각된다. 그 이후 B가 450 근처일 때부터 또 성능이 눈에 띄게 감소하는데 아마 이는 이제 캐시 범위를 벗어나 메모리에도 접근을 해야 되기 때문일 것이다.

**[1. C.]**

이 부분에서는 앞에서 Tiling이 가능하도록 수정한 코드를 thread들이 나눠서 수행할 수 있도록 병렬화를 해야 한다. 기본적인 아이디어는 행렬 c를 같은 크기의 n^2개의 블락으로 나눴을 때 각 블락은 독립적으로 계산할 수 있으므로 한 블락을 한 thread에 할당하여 계산을 수행한다는 것이다. 즉, thread의 개수는 자연수 n에 대하여 n^2이어야 한다는 제한이 있으며, thread의 개수에 맞추어 Tiling을 하면 된다.

먼저 thread의 개수를 argument로 받기 위해 parse\_opt 함수에 해당 부분을 추가하였으며, 이를 전역변수인 tnum에 저장하도록 하였다. main 함수에서 불리는 mat\_mul 함수에서는 tnum의 제곱근인 tsqr변수의 값을 계산하고, 이를 이용하여 Tiling의 블락 크기를 정해 bsize 변수에 저장한다. tsqr와 bsize는 후에 각 thread가 실행하게 되는mat\_mul\_sub 함수 내부에서 각 thread가 접근할 행렬 내의 위치를 정하는데 사용된다. 변수들을 계산한 이후에는 pthread\_attr 타입 변수인 attr 변수를 할당하고, join이 가능하도록 설정한 뒤 thread들을 생성하여 mat\_mul\_sub 함수를 수행하도록 한다. 그 후에 mat\_mul 함수에서는 생성된 thread들이 끝날 때까지 기다리게 되며, 생성된 thread들은 mat\_mul\_sub 함수를 실행하여 행렬 곱셈을 수행한다. mat\_mul\_sub 함수에서 중요한 것은 ii와 jj 변수들의 시작값과 끝나는 조건들인데, 이를 예를 들어 설명하면 다음과 같다. 만약 thread가 9개 있다면, 행렬 c는 다음과 같이 9개의 동일한 크기의 블락으로 나뉘어지며, 각 블락은 하나의 thread가 책임을 지게 된다. 셀 안의 숫자는 thread의 ID를 의미하며, 이 조건에 맞게 ii와 jj의 범위를 정해주어야 한다.

|  |  |  |
| --- | --- | --- |
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

앞서 말했듯이 thread의 개수는 n^2이어야 한다는 제한 때문에 thread의 개수를 1, 4, 9, 16, 25, …, 100이 되도록 실험을 하였으며, 행렬의 NDIM은 2048로 설정하였다. 실험은 천둥 서버에서 thorq를 이용하여 수행하였다. 먼저 전체 문제 크기는 일정할 때 thread의 개수에 따라 성능이 증가하는 strong scalability를 그래프로 표현하면 다음과 같다. (자세한 값들은 matmul\_result.xlsx의 SS탭을 참조하면 알 수 있다) 다음 그래프에서는 결과를 잘 알아볼 수 있도록 x축은 1에서 100까지 1의 간격을 두도록 설정하였다.

**<그림 1-9. Strong Scalability: Thread 개수에 따른 실행시간과 속도>**

위 그래프를 보면 병렬화 오버헤드 때문에 완전히 선형적이지는 않지만, 어느정도는 thread의 개수가 증가함에 따라 성능도 선형적으로 증가한다고 볼 수 있다. 하지만 thread의 개수가 일정 threshold를 넘어서면 성능에 거의 변화가 없는 것을 볼 수 있는데, 이는 thread를 실행시킬 수 있는 CPU 코어의 개수가 한정되어 있기 때문이다. 하지만 천둥 계산 노드 하나에서는 16개의 CPU 뿐이 없는데 왜 thread의 개수가 16개를 넘어가도 성능은 더 높아지는 것일까? 이는 thread의 개수가 CPU 코어의 수보다 많아지더라도 특정 thread들이 긴 latency를 가지거나 I/O 작업 등에 이해 block이 될 때 다른 thread들을 대신 수행할 수 있기 때문이다. 일정 thread 개수 이상을 넘어설 경우 항상 모든 코어에서 thread들이 실행되고 있으므로 더 이상 성능 향상이 없는 것이다.

이제 각 thread가 계산하는 문제의 크기를 일정하게 유지하고 thread의 개수를 증가시켜 weak scalability를 확인해 보자. (이 역시 자세한 값들은 matmul\_result.xlsx의 WS 탭을 참조하면 된다)

**<그림 1-10. Weak Scalability: NDIM과 Thread 개수에 따른 실행시간과 속도>**

Weak scalability를 확인할 때에는 NDIM과 Thread의 개수를 동시에 증가시켜야 하며, 각 thread가 담당하는 sub-matrix의 크기는 일정해야 한다. NDIM을 256의 배수로 증가시켜가며 실험하였으며, thread의 개수는 NDIM=256일 때 1, NDIM=512일 때 4, … NDIM=x\*256일 때 x^2이 되도록 하였다. 그래프를 보면 NDIM과 thread의 개수가 증가함에 따라 성능이 일정하게 유지되지 않고 감소하는 경향을 보이는데, 이는 병렬화 오버헤드로 인한 효과가 점점 커지기 때문이다. 또한 일정 thread 개수 이상이 되면 CPU 코어들이 항상 fully operational이므로 thread의 개수의 증가가 더이상 성능에 영향을 미치지 않게 되며, 이 경우 thread 개수의 증가는 성능 악화만 더 불러오게 된다.

**2.**