M3239.005400 데이터사이언스를 위한 컴퓨팅 2 (001)

Homework #3

Due : 2024/10/30 (Wed)

2024-28413 이정현

1. Matrix Multiplication using OpenMP

-병렬화 방식에 대한 설명

이번 과제에서는 3가지 병렬화 방법을 적용해보았다. (과제2와 거의 동일)

(1) Cache Blocking

효율적인 행렬 곱셈을 위해서, 행렬을 작은 BLOCK으로 나눠서 처리하는 방법을 사용했다. (BLOCK_SIZE는 128로 설정) 이 방법은 memory access pattern이 더 좋아지고, 이에 따른 cache miss가 줄어들기 때문에 프로그램의 성능에 많은 영향을 준다. 행렬 곱셈이 시작되는 가장 바깥쪽 for loop위에 #pragma omp for 을 작성해서 큰 행렬을 여러개의 thread들이 나눠서 병렬로 작업할 수 있도록 해주었다.

(2) Accumulation buffer for results

위에서 사용한 cache blocking의 효과를 더욱 극대화하기 위해서 [BLOCK_SIZE * BLOCK_SIZE] 크기의 buffer를 사용해서 부분 곱셈 결과를 저장했다가 buffer단위로 곱셈 결과를 저장하는 C행렬에 업데이트 해나가는 방법을 선택하였다. 매 곱셈마다 C행렬에 접근하는 것이 아닌, buffer 단위로 접근하게 되기 때문에 C행렬의 memory access를 효과적으로 할 수 있다는 큰 장점이 있다.

여기서 중요한 포인트는 buffer를 #pragma omp parallel 블록 안에서 선언해주었기 때문에 각 thread들은 고유한 buffer를 가질 수 있다. 이로써 얻을 수 있는 효과는 thread 간의 경쟁, 경합이나 데이터 충돌을 피할 수 있다.

(3) Loop Reordering

실제 블록단위 곱셈결과가 buffer에 작성되는 과정에서 일반적인 i,j,k순서가 아닌 i,k,j 순서로 루프를 배치하였다. 이 순서를 이용하면 가장 안쪽의 루프가 반복될 때 A행렬이 reference 되는 위치는 바뀌지 않는다는 것을 확인할 수 있다. (A[i*K + k] 는 j루프에 영향받지 않음)

또한, B행렬도 j 루프 안에서 row를 따라서 연속적으로 access 되고있는 것을 확인할 수 있다. 이 순서대로 짜여진 코드는 Cache miss가 줄어들고, 더욱 효율적이다.

또한, 가장 inner for loop인 j는 항상 0에서 시작해서 BLOCK_SIZE만큼 돈다는 것을 확인할 수 있는데, 이런 패턴은 cache utilization 극대화에 도움을 준다.

```
#pragma omp parallel num_threads(num_threads) //set the number of threads to use
{
  float temp_buff[BLOCK_SIZE*BLOCK_SIZE]; //make temp buffer inside pragma

  #pragma omp for schedule(static)
  for(int ii=0;ii<M; ii+=BLOCK_SIZE){
    for(int jj=0; jj<N; jj+=BLOCK_SIZE){
      memset(temp_buff, 0, sizeof(temp_buff)); //set memory for buffer
      for(int kk=0; kk<K; kk+=BLOCK_SIZE){
```

(1), (2) 에 대한 코드 일부분

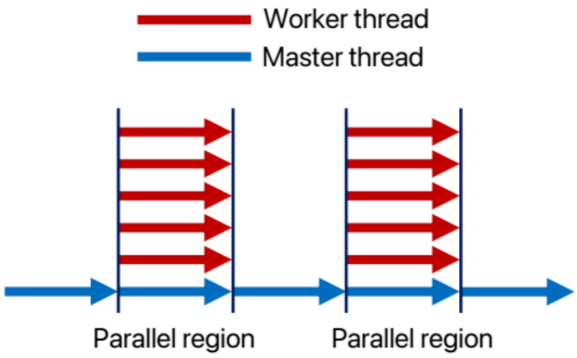
```
for(int i=ii; i<max_i; i++){
  for(int k=kk; k<max_k; k++){
    for(int j=0; j<max_j-jj; j++){ //for cache locality, j should start from 0
      temp_buff[(i-ii)*BLOCK_SIZE + j] += A[i*K + k]*B[k*N + (jj+j)];
    }
  }
}
```

(3)에 대한 코드 일부분

-OpenMP에서 thread 생성 방법, 그리고 컴파일러와 런타임 각각의 역할

OpenMP에서 thread를 생성하기 위해서는 명시적인 생성 함수의 호출이 필요하지 않고, 컴파일러와 런타임 시스템이 자동으로 처리를 해준다.

OpenMP는 fork-join parallelism을 기반으로 작동한다. Fork 과정에서는 master thread가 thread 다발을 spawn하고, join 과정에서는 thread 다발들이 실행을 모두 완료하면 동시에 종료되어 다시 master thread만 남게 된다. OpenMP 프로그램은 처음 하나의 프로세스로 시작하고 (master thread), 병렬 구간에 도달하면 thread다발이 spawn되는 방식으로 작동한다.



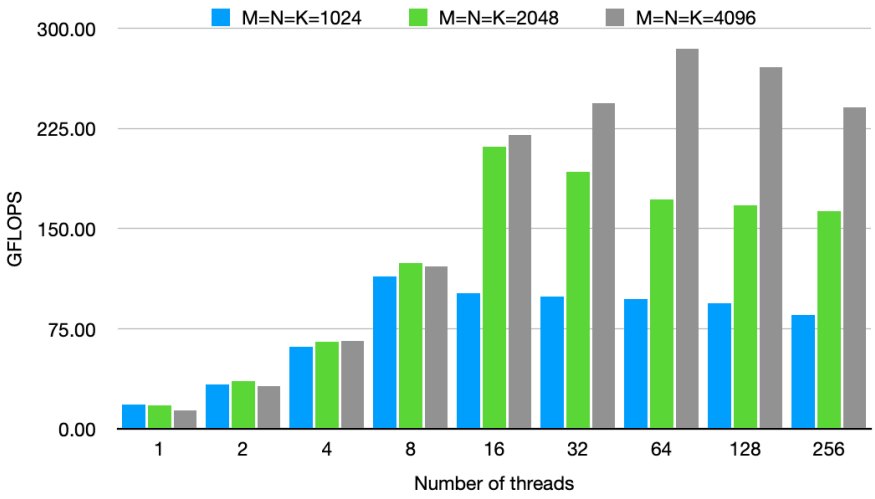
컴파일러 : 병렬 처리를 위한 코드 변경 담당

OpenMP가 사용된 코드를 컴파일하면, 컴파일러는 먼저 #pragma 를 찾아 해당 코드 블록을 병렬로 실행할 수 있도록 코드를 변환해준다. 이 과정에서 컴파일러는 코드 블록에 포함된 변수들을 thread가 안전하게 사용할 수 있도록 scope(shared, private)등을 정리하면서, 코드가 여러개의 thread에서 실행될 수 있도록 준비시켜준다. #pragma 안 parallel이나 for같은 명령어를 발견하면 컴파일러는 자동으로 runtime library를 호출하게 된다. 이는 병렬 실행 요청을 하는 것으로 이해할 수 있다. 이때, thread의 개수 같은 경우는 omp_set_num_threads와 같은 OpenMP API를 이용해서 설정한 후, 런타임에 넘겨준다.

런타임 시스템 : thread의 생성, 작업 분배 및 동기화 관리

OpenMP에서는 프로그램이 시작할 때 thread pool을 생성하게 된다. 병렬로 처리해야하는 코드 블록을 만났을 경우, 이 pool안에 있는 thread들을 활성화 시켜서 사용하게 된다. 병렬 처리가 필요한 코드에서 각 thread가 작업할 양은 런타임 시스템이 분배해주게 되는데, static, dynamic, guided 등의 scheduling 방법으로 이루어질 수 있다. 또한, 런타임은 동기화 관리를 담당하는데, 이는 모든 thread가 병렬 코드 블록에서 작업을 종료할때까지 기다리게 한다. 이후, thread의 사용이 끝나면 해당 thread는 thread pool로 돌아가게 된다.

-스레드를 1개부터 256개까지 사용했을때의 행렬곱 성능 측정



첫 실험으로는 thread의 개수를 2배씩 늘리면서 3가지 크기의 행렬에 대한 곱셈 성능을 측정해보았다. (BLOCK_SIZE는 128로 고정)

1024x1024 크기의 행렬에 대해서는 thread 개수가 16개에서 GFLOPS 값이 peak를 찍은 후 감소하는 양상을 보였다.

더 큰 크기의 행렬(2048, 4096)에 대해서는 thread 개수가 32일때 GFLOPS 성능이 peak를 도달 후, 다소 감소했다는 점을 확인할 수 있었다.

우선 분석에 들어가기 앞서서, 예상했던 결과는 logical core 개수인 64개의 thread를 이용했을 경우에 모두 peak performance를 목격할 수 있을 것이라고 생각하였다. 결론적으로는 큰 행렬(4096)인 경우에만 그러했고, 나머지는 **행렬곱 성능의 peak가 왜 모두 thread 개수가 64개 보다 적은 시점에 일어났는지**에 대한 설명을 하고자 한다. 우리가 실험에 사용하는 계산노드는 64개의 logical core을 가지고 있기 때문에, thread가 64개까지 늘어날 때, 지속적인 성능 향상이 일어날 것이라고 예측했었다. (Thread 개수가 core 개수보다 적기 때문에 경쟁없이 배정 가능)

물론 logical core 64개를 사용하게 해주는 방법인 hyperthreading이 항상 높은 성능 향상을 보여주지는 못한다. 2개의 logical core들이 하나의 physical core에 들어있는 execution unit, cache, memory bandwidth들을 사용해야하는 상황이기 때문에 task에 따라서 overhead가 충분히 발생할 수 있는 것은 사실이다.

세 가지 정도의 가설을 세울 수 있었다. 그 중 두 가지 가설은 모두 OpenMP와 pthread의 작동 방식에서 비롯된 가설이다. **첫째**, 조사 결과 OpenMP가 항상 hyperthread 효과를 최대로 이용하지 않을수도 있다는 부분이었다. 컴파일러와 환경에 따라서 설정을 해주어야하는 부분일수도 있다라는 사실을 알게 되었고 (OMP_PROC_BIND 와 같은 명령어를 통해), 이것이 실험 결과가 보여주는 원인 중 하나일수도 있겠다는 생각을 하였다. 그러나 컴파일하는 과정에서 run.sh 파일 안에 [numactl —physcpubind 0-63] 명령어가 포함되어 있는 것을 확인하였고, 64개의 core모두 사용하고 있다는 것을 확인하였다.

둘째, pthread와 다르게 OpenMP는 thread pool 에서 thread들을 spawn 후 재사용하는 구조를 가지고 있는데, 여기서 synchronization overhead가 크게 작용했을 수도 있다는 점이었다. Thread의 개수가 많아질수록 이 문제는 더 대두될 가능성이 있으며, 이러한 이유로 32개 이후로는 오히려 성능 저하가 나타났다는 가능성을 염두에 두었다. 그리고 마지막 **셋째,** 64개 보다 적은 thread들을 사용한 상태에서 이미 cache utilization이나 memory bandwidth utilization이 극대화가 되어있다는 생각을 하였다. (아래에 추가 설명)

<실험 결과 그래프 분석>

작은 크기의 행렬(1024)의 같은 경우, 많은 비중의 데이터가 L2 (혹은 L3) cache에 들어갈 수 있었던 것으로 판단된다. 이는 memory latency를 최대한 줄이면서 성능 향상을 가능하게 해준다. 그러나 더 많은 thread들을 사용할수록 cache contention이 늘어난다. 8개일때 peak performance를 보여준 결과를 바탕으로, thread가 8개 일때 이미 cache usage가 최적화되어 있는 것으로 보인다. 이 이상 thread를 늘리면 오히려 cache thrashing (여러개의 thread가 서로의 데이터를 shared cache에서 evict함) 현상이 일어나서 성능이 감소할 여지가 있다.

또한, 작은 행렬의 경우, 하나의 thread가 할당받은 양은 줄어들지만 synchronization overhead는 일정하게 유지된다. Thread의 개수가 너무 많아지면 thread management, scheduling, synchronization에서 오는 overhead가 너무 커져버리는 상황이 발생해서 병렬화의 효과가 감소한다.

큰 크기의 행렬(4096) 같은 경우, 데이터들을 cache 보다는 main memory에서 가져와야 한다. 이 경우에는 cache의 효과는 다소 감소할 수 있다. 그러나 logical core 개수인 64개의 thread까지 성능이 증가하는 이유는 여러개의 core에 걸친 memory bandwidth utilization에서 오는 이점이 computational overhead보다 우세했기 때문이다.

중간 크기 행렬(2048)의 경우 위에서 설명한 두 상황의 중간 어디쯤에서 peak performance를 낼 수 있는 지점은 32개의 thread 사용으로 보여진다.

요약하면, 작은 행렬과 큰 행렬의 곱셈 성능 실험결과에서 알 수 있는 사실은 parallelization overhead와 computational workload 사이의 trade-off이다. 작은 행렬 곱셈에서는 OpenMP thread들을 관리하는 과정(synchronization, scheduling)에서 발생하는 overhead가 더 크고, 대략 8개의 thread를 사용할 때 병렬화와 overhead의 최적 균형이 맞춰진다고 볼 수 있다. 큰 행렬 곱셈에서는 computational workload가 더 큼으로, parallelization overhead가 상대적으로 덜 중요해진다. 이로인해 64개 thread를 사용할때까지 성능증가를 확인할 수 있는 것이다.

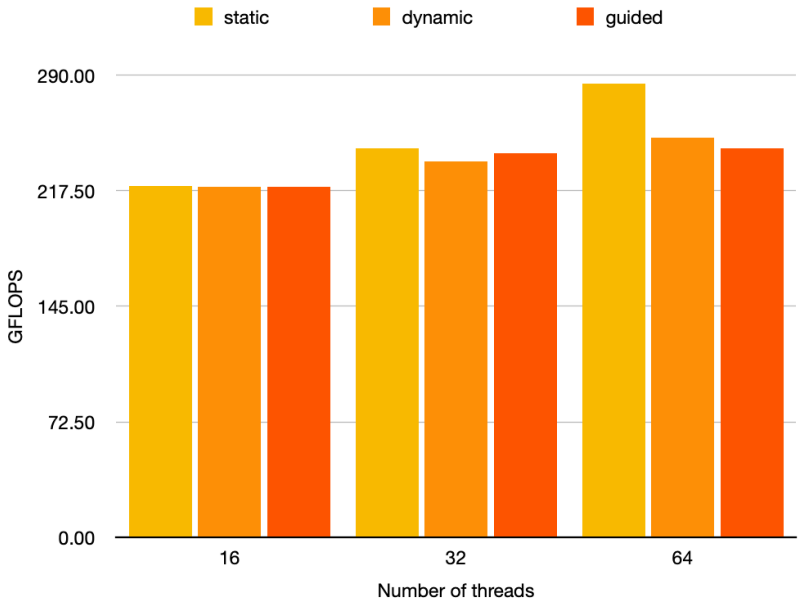
-OpenMP의 loop scheduling 방법, 그리고 static, dynamic, guided의 방식 비교

Static : 이 방법은 컴파일 타임에 loop iterations들이 동일한 크기의 chunk들로 나누어지게 된다. 각각의 chunk들은 루프가 시작하기 전에 thread를 배정받게 된다. 모든 iteration들이 비슷한 양의 computation workload라면 성능이 가장 잘 나오는 방법이다. 계산량이 크게 차이나는 상황이라면 thread마다 load imbalance 문제가 큰 문제로 작용해서 성능 저하가 나타날 수 있다.

Dynamic : 이 방법은 execution 도중에 iteration들이 thread를 배정받게 된다. 어떤 thread가 본인이 배정받은 chunk를 모두 처리했을때, 곧바로 다른 chunk를 요청해서 새로 할당받게 된다. iteration마다 workload가 크게 차이날때 사용하면 좋은 방법이다. Load imbalance 문제를 해결해주지만, synchronization 과 scheduling overhead로 인해서 실행시간은 다소 오래 걸릴 수 있다는 단점이 있다.

Guided : 이 방법도 dynamic scheduling 방법이지만, 특징은 chunk의 크기가 크게 시작해서 iteration이 진행될수록 점점 dynamic하게 감소한다는 점이다. 초기 iteration들에는 계산량이 많지만, 뒤로 갈수록 감소하는 패턴의 workload를 다루고자 할때 사용하면 유용한 방법이다. Scheduling overhead를 가지고 있지만 dynamic 방법보다는 적으며, load imbalance 문제를 효율적으로 해결해준다.

아래 그래프는 4096 크기 행렬 곱셈에 대해 세 가지 scheduling 방법을 적용해본 결과를 나타내는 그래프이다.



16, 32, 64개의 thread를 사용하는 경우에 대해 각각 실험을 하였다. 이번 for loop의 각 iteration이 담당하는 계산량은 매우 균일했던 관계로, 엄청 유의미한 차이는 찾아볼 수 없었다. 적은 개수의 thread를 사용하는 경우, 세 scheduling 방법 모두 비슷한 결과를 주었다. 64개 정도의 thread를 사용했을때는 static과 나머지 두 방법 사이에 차이가 조금 나타났는데, 이는 thread가 많아짐으로서 동적 scheduling의 overhead가 살짝 드러난 것으로 유추 해볼 수 있다.

-정확성 : ./run_validation.sh 10개 모두 VALID

-성능 : ./run_performance.sh 실행결과 최대값 373GFLOPS

```
shpc106@login3:~/skeleton/hw3/openmp$ ./run_performance.sh
srun: job 890795 queued and waiting for resources
srun: job 890795 has been allocated resources
Options:
  Problem size: M = 4096, N = 4096, K = 4096
  Number of threads: 32
  Number of iterations: 10
  Print matrix: off
  Validation: on

Initializing... done!
Calculating...(iter=0) 0.498171 sec
Calculating...(iter=1) 0.380652 sec
Calculating...(iter=2) 0.328185 sec
Calculating...(iter=3) 0.328435 sec
Calculating...(iter=4) 0.328276 sec
Calculating...(iter=5) 0.328361 sec
Calculating...(iter=6) 0.458531 sec
Calculating...(iter=7) 0.375020 sec
Calculating...(iter=8) 0.328293 sec
Calculating...(iter=9) 0.328630 sec
Validating...
Result: VALID
Avg. time: 0.368255 sec
Avg. throughput: 373.216434 GFLOPS
```

2. Estimating Cache Size

공통적으로 workload의 크기가 cache의 용량을 초과하는 순간, 성능(실행시간)이 급격하게 저하될것이라는 현상을 이용해서 각 cache크기를 유추해보고자 하였다. (모든 실험 thread 1개 사용)

최적화 과정에서 thread마다 사용하는 BLOCK_SIZE*BLOCK_SIZE 크기의 버퍼도 cache안에 확실히 들어갈 수 있도록 BLOCK_SIZE를 64로 변경한 후, cache 크기 추정 실험을 진행하였다.

또한, run.sh 파일안에 —physcpubind 0으로 수정한 후, 실험을 진행하였다.

-L1 cache

L1 cache는 보통 크기가 매우 작은 것이 특징이다. 따라서 작은 크기의 행렬 (32x32) 부터 시작해서 크기를 1씩 증가시켜보았다. 행렬크기 38과 39사이에서 다소 실행시간의 증가를 목격하였고, 이 범위 사이에 L1 cache의 크기가 있을 것이라고 생각하게 되었다.

L1 cache는 크기가 매우 작은 관계로 별다른 방법은 사용하지 않고 크기를 1씩 늘리면서 변화 구간을 추적하였다. 아래 표는 탐색의 결과이다.

Estimating L1 cache

Matrix size (M=K=N)	Avg. time
32	0.000020
33	0.000022
34	0.000027
35	0.000032
36	0.000037
37	0.000041
38	0.000040
39	0.000051
40	0.000039

A,B,C 행렬 크기가 38x38에서 39x39로 넘어가는 순간 급격한 실행시간의 변화를 목격할 수 있었다. (빨간 글씨)

이번 과제에서 구현한 matmul 코드에는 행렬 3개와 더불어 [BLOCK_SIZE*BLOCK_SIZE] 크기의 버퍼를 사용하였다. 이들의 자료형은 모두 float (4byte) 임을 고려해서 L1 cache 크기를 추정한 결과는 다음과 같다.

-A, B, C 행렬 크기 합 = $3 \times 38^2 \times 4$ = 17,328 bytes

-Buffer 크기 = $64^2 \times 4$ = 16,384 bytes

-L1 cache 추정 크기 = 33,712 bytes = **33.712 KB**

Private 유무를 확인하기 위해서는 두 개 thread를 다른 코어에서 동시에 실행해보는 방법이 있다. 각 thread가 서로 다른 데이터를 처리하게 한 후(ex. 서로 다른 행렬곱), 성능을 각각 측정했을 때 차이가 거의 없다면thread들이 서로 독립적으로 실행된다는 것이고, L1 cache가 private하다고 결론내릴 수 있다.

만약 shared 라면, cache contention으로 인한 성능 저하가 나타날 수 있다. 보통 L1 cache들은 private으로 만들어지기 때문에 계산노드에 장착된 CPU의 L1 cache는 private이라고 생각한다.

-L2 cache

L2 cache는 L1 cache 보다는 크지만, 속도가 느린것이 특징이다. 행렬의 크기를 L1 cache를 초과하는 수준으로 설정하고, 이후 실험과정은 거의 유사하게 진행하였다. 또다시 실행시간의 성능 급감 지점을 찾고자 하였고, 296x296에서 297x297로 행렬의 크기가 커지는 지점에서 성능 감소를 목격하였다. 아래 표는 탐색의 결과이다. 효율적인 탐색을 위해서 큰 범위 내에서는 이진탐색을 이용해보았다.(노란색 칸이 이진탐색 행렬크기를 의미한다)

Estimating L2 cache

Matrix size (M=K=N)	Avg. time
256	0.002736
288	0.003317
290	0.003519
296	0.003564
297	0.004113
300	0.004223
320	0.004406

A,B,C 행렬 크기가 296x296에서 297x297로 넘어가는 순간 급격한 실행시간의 변화를 목격할 수 있었다. (빨간 글씨)

L2 cache의 크기를 추정하기 위해, 296x296 크기의 행렬까지 L2 cache 안에 들어온다고 가정했을때 cache 크기를 구해본다.

-A, B, C 행렬 크기 합 = $3 \times 296^2 \times 4$ = 1,051,392 bytes

-Buffer 크기 = $64^2 \times 4$ = 16,384 bytes

-L2 까지 전체 cache 추정 크기 = 1,067,776 bytes = 1,067.776 KB ≈ **1.07 MB**

위에서 구한 크기는 L1과 L2 cache 크기가 합쳐진 값이므로, 앞서 구한 L1 크기를 빼주면 L2 cache 크기를 추정해볼수 있다. 1,067.776 KB - 33.712 KB = 1,034.064 KB ≈ **1.03 MB**

L2 cache의 private 여부를 확인하기 위해서는 L1 cache와 마찬가지로 두 개의 thread를 다른 코어에서 서로 다른 데이터에 대해 동시에 실행해보는 방법이 있다. 성능 저하가 없다면, 계산노드의 L2 cache는 private이라고 결론내릴 수 있다.

Private 이기 때문에, 코어가 늘어날수록 L2 cache 크기도 늘어난다고 생각할 수 있고, 앞서 1번 문제에서 측정한 행렬곱의 성능 중 1024x1024 행렬은 thread개수가 8개일때 모든 코어에 걸쳐 L2 cache에 전부 들어가서 peak performance를 보여주는 것으로 예상된다. 그 이후로도 thread가 늘어나도 성능이 계속 줄어들지 않은 이유도 L2 cache의 private 특성에서 오는 것으로 생각해볼수 있다. (Cache contention이 없기 때문)

-L3 cache

L3 cache는 보통 shared되는 cache 이고, 이 cache가 완전히 차면 실행 시간의 성능 저하가 일어날 것이라고 예상할 수 있다. 위 두개의 실험과 비슷한 과정으로 L3 cache의 크기를 추정해보았다. 아래 표는 탐색의 결과이다. (노란색 칸이 이진탐색 행렬크기를 의미한다)

Estimating L3 cache

Matrix size (M=K=N)	Avg. time
1024	0.126666
1536	0.435693
1792	0.705631
1920	0.770281
1984	0.827504
1985	0.931252
1986	0.942339
2016	1.114254
2048	1.141833

A,B,C 행렬 크기가 1984x1984에서 1985x1985로 넘어가는 순간 급격한 실행시간의 변화를 목격할 수 있었다. (빨간 글씨)

L3 cache의 크기를 추정하기 위해, 1984x1984 크기의 행렬까지 L3 cache 안에 들어온다고 가정했을때 cache 크기를 구해본다.

-A, B, C 행렬 크기 합 = $3 \times 1984^2 \times 4$ = 47,235,072 bytes

-Buffer 크기 = $64^2 \times 4$ = 16,384 bytes

-L3 까지 전체 cache 추정 크기 = 47,251,456 bytes = 47,251.456 KB ≈ **47.25 MB**

위에서 구한 크기는 L1, L2, L3 cache 크기가 합쳐진 값이므로, 앞서 구한 L1과 L2 cache 크기를 빼주면 L3 cache 크기를 추정해볼수 있다. 47.25 - (1.03 + 0.337) ≈ **45.88 MB**

L3 cache의 shared 유무를 확인하기 위해서는 여러개의 코어가 동시에 접근할 때 성능저하가 생기는지를 알아보면 된다. 앞서 1번 문제에서 행렬곱 성능을 측정했을때, (여러 코어를 이용하면) L3 안에 들어갈 것으로 예상되는 2048x2048 크기의 행렬을 곱할 때 16개 이상의 thread를 사용하면 성능이 오히려 떨어지는 것을 확인하였다. 이는 여러개의 코어(thread)가 하나의 L3 cache를 공유하기 때문에 발생하는 성능 간섭으로 생각해볼수있다. 따라서 L3 cache는 shared로 결론내릴 수 있다.

-추정값과 실제값 비교

계산 노드의 실제값은 우선 [srun -N 1 lscpu] 명령어를 통해서 확인하였다.

```
L1d cache:                       1 MiB
L1i cache:                       1 MiB
L2 cache:                        32 MiB
L3 cache:                        44 MiB
```

(화면에 표시되는 단위가 MiB (mebibyte) 였는데, 1 MiB 는 1.049 MB와 거의 동일한 값으로 잠시 무시하기로 한다.)

인터넷을 통해서 찾은 계산노드의 CPU - Intel(R) Xeon(R) Silver 4216 @ 2.10GHz의 cache 크기는 각각 32KB, 1MB, 22MB 였다.

L1 cache는 private이고, 각 코어마다 존재한다. 계산노드에는 16 코어 CPU가 2개 있으므로 **32KB * 32 = 1MB** 값을 확인할 수 있었다.

L2 cache도 private이고, 각 코어마다 존재한다. 계산노드에는 16 코어 CPU가 2개 있으므로 **1MB * 32 = 32MB** 값을 확인할 수 있었다.

L3 cache는 shared고, 모든 코어들이 공유한다. 계산노드에는 CPU가 2개 있으므로 **22MB * 2 = 44MB** 값을 확인할 수 있었다.

이로써 위에서 추측한 L1, L2는 private. L3는 shared 일 것이라는 결론이 맞음을 확인했다.

아래 표는 실제값(MB 로 변환)과 추정값을 정리해보았다. (Per core 추정값)

Cache	Estimated (MB)	Actual (MB)	Difference (%)
L1	0.337	0.328	2.74%
L2	1.03	1.05	1.90%
L3	45.88	46.14	0.56%

-3개의 cache 모두 다 생각보다 작은 오차로 추정하는데에 성공하였다. 용량이 매우 작은 L1 cache에서는 작은 오차가 더 큰 비율로 나타나게 되었다. 전반적으로는 합리적인 결과를 얻은 실험이였던것 같다. L1은 용량도 작고 속도도 매우 빠르기 때문에 memory access pattern이 오차를 만들어낸 큰 원인이라고 생각하였다.

-이 실험에서 64*64 크기의 버퍼를 선언하고 사용하였는데, cache 크기를 추정할때 이 부분을 고려했지만 미처 생각하지 못한 부분에서 cache memory를 추가적으로 잡고있거나 할 가능성이 있을수도 있다. 이는 cache의 underestimation/overestimation을 일으켰을 수 있다. (특히 용량이 매우 작은 L1에서 더 가능성이 높다)

-최근 사용되는 CPU들은 prefetching을 많이 한다고 알고있다. 이로 인해 필요한 것 보다 조금 더 많은 데이터가 cache에 들어올수도 있고, 여기서 측정 오차가 발생했을 가능성도 존재한다.

-L3 cache는 shared cache이기 때문에 thread들의 동시 접근에서 발생하는 overhead등이 약간의 오차를 만들어냈다고 생각할 수 있다.