HW3: Matrix Multiplication

Giwon, Seo
Dept. of Computer Science. Yonsei University
2015147531 / mgp10029

1. 과제 요약

이번 과제의 목적은 행렬 곱셈의 실행 시간을 단축하는 것이다. 행렬 곱셈은 기본적으로 A의 행과 B의 열을 곱하여 C의 한 원소를 계산하는 것이기 때문에 $O(n^3)$의 시간복잡도를 가지게 된다. 하지만 컴퓨터의 구조상 캐시에서의 메모리 접근이 우선시되기 때문에 행렬의 크기가 캐시의 크기보다 크다면 실질적인 시간은 이보다 더 걸리게 된다. 따라서, '병렬 처리'와 더불어 캐시 메모리 크기에 맞게 계산 방법을 고안하여 행렬 곱셈의 실행 시간을 효과적으로 단축해 보았다.

2. 서론

기본적으로 병렬 처리에서 mutex의 사용은 많은 시간을 소요하기 때문에 곱셈의 대상이기 때문에 같은 부분의 메모리 접근이 잦은 A와 B에 대해 병렬 처리를 하기보다는 C에 대해 단순한 반복문의 인덱스를 조절하는 방법으로 병렬 처리를 진행하는 것이 효과적이다.

3. 사용 기법

3-1. 병렬 처리

행렬 C의 한 원소에 대해 행렬 A와 행렬 B의 원소 메모리 read만 일어나고 write는 행렬 C에 적용되므로 행렬 C의 인덱스에 대하여 병렬 처리를 활용하였다. openmp 라이브러리를 활용하여 쉽게 구현하였다.

3-2. 행렬 Transpose

이번 과제에서 행렬은 배열에 저장되어 있다. 실제 메모리상에서 배열은 연속적인 주소 값을 가지기 때문에 배열 안의 한 원소에 대한 메모리 접근을 할 때, 캐시에 해당 인덱스부터 캐시 라인 크기만큼의 정보를 모두 가져와 저장하게 된다. 이는 첫 번째 피연산자인 A의 경우 행에 해당하는 원소를 불러오기 때문에 이와 관계없지만 열에 해당하는 원소를 불러와 계산해야 하는 두 번째 피연산자인 B의 경우 계산 과정에서 A보다 많은 수의 캐시 미스를 발생하게 된다. 캐시 미스에 따른 메모리 접근은 더 많은 실행 시간을 가지게 된다. 이를 해결하기 위해 두 번째 피연산자인 행렬 B를 행과 열을 바꾸는 작업을 통해 필연적으로 발생하는 캐시 미스를 일부분 완화했다.

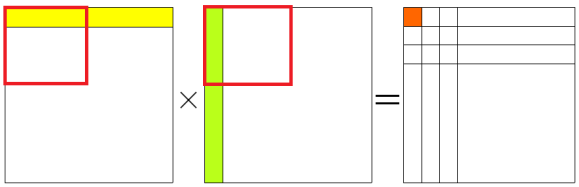

그림 1 . $C = \sum_{j=0}^{n} \sum_{i=0}^{n} r_i(A) \times c_j(B)$

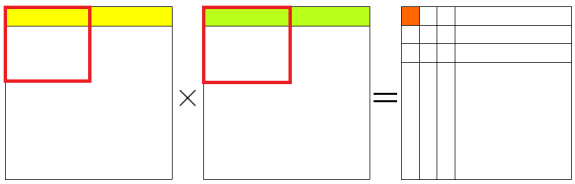

그림 2 . $C = \sum_{i=0}^{n} \sum_{j=0}^{n} r_i(A) \times r_j(B^T)$

3-2. 행렬 분할

전치 행렬을 통해 B의 열에 접근하면서 발생하는 캐시 미스를 줄여주면서 캐시를 통한 계산이 메모리에 직접 접근하면서 계산하는 방법보다 현저히 빠르다는 것을 알 수 있었다. 이로 인해 행렬 A, B, C 모두를 캐시에 저장할 수 있다면 더욱 빠른 계산이 가능하다는 것을 추측할 수 있다. 캐시 메모리의 크기가 4096 × 4096인 이번 과제의 input 행렬 3개의 크기보다 크다면 문제가 되지 않지만, 주어진 L1 캐시의 크기는 32KB로 4 Byte 크기의 정수가 16개 들어가는 캐시 라인과 함께 단순 계산을 통해 16 × 16의 작은 행렬 블록이 32개 들어갈 수 있다는 것을 알 수 있다. 앞선 그림 1과 그림 2를 통해 블록별 계산을 통해 C를 충분히 계산할 수 있다는 것을 알 수 있다.

3-3. Loop reordering

행렬에 접근하는 원소의 순서에 따라 시간적 차이를 보일 수 있다. 따라서 matrixC를 계산하는 과정

에서 블록의 위치를 특정하는 변수 bi, bj와 블록 내에서의 인덱스를 특정하는 변수 i, j의 순서를 임의로 바꾸어 시간적 이점을 얻고자 하였다.

4. 코드

4-1. 행렬 분할 부분

```
int* transB = new int[n * n];
#pragma omp parallel for
for (int i=0; i<n; i++)
for (int j=0; j<n; j++) {
  transB[i*n + j] = matrixB[j*n + i];
}
```

그림 3 . 초기 전치 행렬 구하는 방식

matrixB가 const 변수로 설정되어 있기 때문에 B의 전치 행렬을 저장하기 위한 transB 배열을 새롭게 할당한다. 초기 방법으로 단순히 행과 열을 바꾸는 방식의 2중 for문으로 구성하여 전치 행렬을 구하는 방식을 채택하였다.

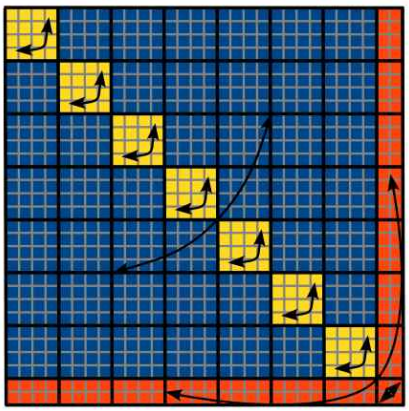

그림 4. multithreading방식

이후 캐시 메모리를 활용하는 이번 과제의 취지에 맞게 전치 행렬을 구하는 방식 또한 블록으로 나누어 구하는 참고 논문[1]의 방법을 참고하였다. 행렬의 크기가 블록의 크기로 나누어떨어지지 않는 일반적인 경우에 대해 정리를 한 이 방법에서 4096의 약수인 16과 32만을 블록 크기로 설정하여 비교하였기 때문에 그림 4에서 빨간색 부분에 해당하는 swap은 생략하여 구현하였다. 이후 openmp를 활용하여 병렬 처리를 하여 시간적 이점을 얻었다.

4-2. 행렬 분할

```
#pragma omp parallel for
for (int bi = 0; bi < n; bi += blockSize) {
  for (int bj = 0; bj < n; bj += blockSize) {
    for (int bk = 0; bk < n; bk += blockSize) {
      for (int i = 0; i < blockSize; i++) {
        for (int j = 0; j < blockSize; j++) {
          int sum = 0;
          for (int k =0; k < blockSize; k++) {
            sum += matrixA[(bi + i)*n + (bk + k)] * transB[(bj + j)*n + (bk + k)];
          }
          matrixC[(bi + i)*n + (bj + j)] += sum;
```

블록별 접근을 위한 인덱스 bi, bj, bk를 설정하고 블록 내에서의 행렬 원소 접근을 위한 인덱스 i, j, k를 설정하여 차례대로 계산하였다. matrixC의 원소에 대한 메모리 접근을 k에 대한 for문을 진행하면서 항상 하게 되면 시간적 손해를 보게 되어 sum 변수를 새롭게 설정하여 모든 계산을 진행한 후 마지막에 한 번만 접근하여 더해주었다.

5. 결과 분석

5-1. 블록과 타일의 크기

혼동을 피하고자 행렬 B의 전치 행렬을 구할 때 사용한 블록의 크기를 타일의 크기로 바꾸어 명칭하였다.

Block	Tile	Time(sec)
16	16	2.94687
16	32	2.91563
32	16	1.06509
32	32	1.07278

결과를 보았을 때 타일의 크기는 실행 시간 감소에 큰 영향을 주지 못하였지만, 블록의 크기는 실행 시간 감소에 큰 영향을 주었다. 전치 행렬을 구하는 과정보다 곱셈 연산을 실행하는 과정에서 메모리 접근이 많으므로 더욱 큰 실행 시간 차이를 보이는 것으로 보인다.

5-2. 전치 행렬의 사용 여부

	Time(sec)
사용 X	?
사용 O, 초기 방법	1.12674
사용 O, 분할	1.06509

전치 행렬을 사용하지 않는 경우, 서버 내에서 실행 시간이 길어 끝까지 수행하지 못하여 결과를 내지 못하였다. 이를 통해 전치 행렬을 사용하는 것이 캐시에 대한 활용도를 더욱 높여주기 때문에 전치 행렬을 구하는 만큼의 시간적 손해를 감수하고도 계산을 할 가치가 있음을 알 수 있다. 또한, 일

반적인 방법으로 구하는 것보다 캐시에 대한 활용도를 높여 행렬 분할을 사용하여 전치 행렬을 구하는 방법이 더욱 실행 시간을 줄여준다는 것을 알 수 있다.

5-3. Loop reordering

순서	Time(sec)
bi, bj, i, j	1.0189
bi, bj, j, i	1.52093
bj, bi, i, j	1.0544
bj, bi, j, i	1.23424

i와 j의 순서로 실행하는 것이 가장 빠른 결과를 보여주었다. 이에 대한 이유는 행렬 C의 원소에 접근할 때, 행이 달라지는 것보다 열이 달라지면서 for 문을 실행하는 것이 캐시 미스의 빈도를 줄이기 때문으로 추측할 수 있다.

6. 참조

[1] Andrey Vladimirov,「Multithreaded Transposition of Square Matrices with Common Code for Intel Xeon Processors and Intel Xeon Phi Coprocessors」, Colfax Research, 2013.08.12