컴퓨팅2 과제 3

1. Matrix multiplication using openMP
    a) 자신의 병렬화 방식에 대한 설명

```
#pragma omp parallel private(i, j, k, temp, index_a, index_b, index_c) shared(A, B, C)
{
    #pragma omp for schedule(dynamic)
    for (int ii = 0; ii < M; ++ii){
        for(int kk = 0; kk < K; kk += block_K){
            for (int jj = 0; jj < N; jj += block_N){

                for(k = 0; k < block_K; ++k){
                    index_a = ii * K + (kk + k);
                    temp = A[index_a];

                    index_c = ii * N + jj;
                    index_b = (kk + k) * N + jj;

                    for (int j = 0; j < block_N; j+=8){

                        // for(j = 0; j < cache_line; ++j){
                        C[index_c] += temp * B[index_b];
                        C[index_c + 1] += temp * B[index_b + 1];
                        C[index_c + 2] += temp * B[index_b + 2];
                        C[index_c + 3] += temp * B[index_b + 3];

                        C[index_c + 4] += temp * B[index_b + 4];
                        C[index_c + 5] += temp * B[index_b + 5];
                        C[index_c + 6] += temp * B[index_b + 6];
                        C[index_c + 7] += temp * B[index_b + 7];

                        index_c += 8;
                        index_b += 8;

                    }
```

(코드 중 핵심 병렬화 부분만 캡쳐)

Loop unrolling: 덧셈 instruction을 8개로 늘림. Introduing block_N: B의 열연산을 블록 단위로 수행, introducing block_K: B의 행연산(A의 열연산)을 블록 단위로 수행, OpenMP parallelism: A의 열연산에 대해 openMP dynamic scheduling 적용

    b) openMP에서 스레드 생성은 어떤 식으로 이루어지는가?

#pragma omp parallel을 통해서 생성/활성화할 수 있다. openMP 명령어가 들어왔을 때 컴파일러는 스레드 풀을 미리 생성하고, 코드의 안정적인 시행을 위한 장치(예: barrier, 스레드 종료 및 회수) 등을 삽입한다. 런타임 시스템은 parallel 등의 명령어를 만나면 풀을 활성화하고, 스케쥴링 등의 작업을 수행한다.

    c) 스레드를 1개부터 256개까지 사용했을 때의 성능

```
shpc125@elogin3:~/hw3$ srun --nodes=1 --exclusive numactl --physcpubind 0-63 ./main -v -t 1 4096 4096 4096 | grep "Avg. time"
srun: job 902130 queued and waiting for resources
srun: job 902130 has been allocated resources
Avg. time: 38.370778 sec
shpc125@elogin3:~/hw3$ srun --nodes=1 --exclusive numactl --physcpubind 0-63 ./main -v -t 2 4096 4096 4096 | grep "Avg. time"
srun: job 902132 queued and waiting for resources
srun: job 902132 has been allocated resources
Avg. time: 21.706289 sec
shpc125@elogin3:~/hw3$ srun --nodes=1 --exclusive numactl --physcpubind 0-63 ./main -v -t 4 4096 4096 4096 | grep "Avg. time"
srun: job 902134 queued and waiting for resources
srun: job 902134 has been allocated resources
Avg. time: 10.494462 sec
shpc125@elogin3:~/hw3$ srun --nodes=1 --exclusive numactl --physcpubind 0-63 ./main -v -t 8 4096 4096 4096 | grep "Avg. time"
srun: job 902135 queued and waiting for resources
srun: job 902135 has been allocated resources
Avg. time: 5.500549 sec
shpc125@elogin3:~/hw3$ srun --nodes=1 --exclusive numactl --physcpubind 0-63 ./main -v -t 16 4096 4096 4096 | grep "Avg. time"
srun: job 902136 queued and waiting for resources
srun: job 902136 has been allocated resources
Avg. time: 2.862555 sec
shpc125@elogin3:~/hw3$ srun --nodes=1 --exclusive numactl --physcpubind 0-63 ./main -v -t 32 4096 4096 4096 | grep "Avg. time"
srun: job 902137 queued and waiting for resources
srun: job 902137 has been allocated resources
Avg. time: 1.806894 sec
shpc125@elogin3:~/hw3$ srun --nodes=1 --exclusive numactl --physcpubind 0-63 ./main -v -t 64 4096 4096 4096 | grep "Avg. time"
srun: job 902138 queued and waiting for resources
srun: job 902138 has been allocated resources
Avg. time: 1.272905 sec
shpc125@elogin3:~/hw3$ srun --nodes=1 --exclusive numactl --physcpubind 0-63 ./main -v -t 128 4096 4096 4096 | grep "Avg. time"
srun: job 902139 queued and waiting for resources
srun: job 902139 has been allocated resources
Avg. time: 1.929201 sec
shpc125@elogin3:~/hw3$ srun --nodes=1 --exclusive numactl --physcpubind 0-63 ./main -v -t 256 4096 4096 4096 | grep "Avg. time"
srun: job 902140 queued and waiting for resources
srun: job 902140 has been allocated resources
Avg. time: 2.941577 sec
```

스레드 개수가 늘어남에 따라 일정한 추세로 성능이 증가하지 않는다. 38, 22, 10, 6, 3, 2, 1, 2, 3초가 걸린다(반올림). 이는 스레드 간 동기화 및 커뮤니케이션 cost, 스레드 생성, 관리 및 회수 등에 걸리는 시간 때문인 것으로 추정된다.

d) 가장 높은 성능을 보이는 스레드 개수와 상대 연산량

```
shpc125@elogin3:~/hw3$ srun --nodes=1 --exclusive numactl --physcpubind 0-63
./main -v -t 64 4096 4096 4096
srun: job 902180 queued and waiting for resources
srun: job 902180 has been allocated resources
Options:
  Problem size: M = 4096, N = 4096, K = 4096
  Number of threads: 64
  Number of iterations: 1
  Print matrix: off
  Validation: on

Initializing... done!
Calculating...(iter=0) 1.274625 sec
Validating...
Result: VALID
Avg. time: 1.274625 sec
Avg. throughput: 107.826957 GFLOPS
```

가장 높은 성능을 보이는 스레드 개수는 64이고, 이때의 속도는 107.82 GFLOPS이다. 이는 최대 연산량 2150.4GFLOPS 대비 약 5%이다. 이는 AVX 기능을 활용하지 않은 것은 물론이고, SIMD 기능도 활용하지 않았기 때문에 생긴 현상으로 추정된다. 성능을 향상시키기 위해서는 앞의 방법론에 더해 캐시를 고려한 블록 사이즈 수정, temporal and spatial locality를 고려한 캐시 미스 줄이기 등의 방법론을 써 볼 수 있겠다.

e) Loop scheduling, static, dynamic, guided

```
shpc125@elogin3:~/hw3$ gcc -o main main.c matmul.c util.c -l pthread -O3 -fopenmp
shpc125@elogin3:~/hw3$ srun --nodes=1 --exclusive numactl --physcpubind 0-63 ./main -v -t 64 4096 4096 4096 | grep "Avg."
srun: job 902279 queued and waiting for resources
srun: job 902279 has been allocated resources
Avg. time: 1.409345 sec
Avg. throughput: 97.519736 GFLOPS
shpc125@elogin3:~/hw3$ gcc -o main main.c matmul.c util.c -l pthread -O3 -fopenmp
shpc125@elogin3:~/hw3$ srun --nodes=1 --exclusive numactl --physcpubind 0-63 ./main -v -t 64 4096 4096 4096 | grep "Avg."
srun: job 902280 queued and waiting for resources
srun: job 902280 has been allocated resources
Avg. time: 1.269195 sec
Avg. throughput: 108.288254 GFLOPS
shpc125@elogin3:~/hw3$ gcc -o main main.c matmul.c util.c -l pthread -O3 -fopenmp
^[[Ashpc125@elogin3:~/hsrun --nodes=1 --exclusive numactl --physcpubind 0-63 ./main -v -t 64 4096 4096 4096 | grep "Avg."
srun: job 902281 queued and waiting for resources
srun: job 902281 has been allocated resources
Avg. time: 1.370846 sec
Avg. throughput: 100.258466 GFLOPS
```

위에서부터 차례대로 static, dynamic, guided 옵션을 적용했다. Static은 스레드별 고정 개수의 loop 연산을 수행하는 것이고, dynamic은 일찍 끝난 스레드에 남아있는 고정 개수의 loop 연산을 동적으로 할당하는 것이고, guided는 많은 개수를 할당하고 남은 loop 연산의 수에 따라 개수를 점점 줄여 할당하는 것이다. dynamic보다 오버헤드가 적다. 실험을 통해 확인해 본 결과, 확실히 static보다는 dynamic 혹은 guided가 빠르다. 이 경우에서는 M=N=K=4096으로 실험했기에 dynamic이 더 빨랐을 수도 있으나, 다른 경우로 실험한다면 guided가 더 효율적일 수도 있다.

2. Estimating cache size

방법론: 1의 코드보다 단순한 코드로 실험을 진행하였다.

```
#pragma omp parallel private(i, j, k, temp, index_a, index_b, index_c) shared(A, B, C)
{
    #pragma omp for
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; k < K; k++) {
                C[i * N + j] += A[i * K + k] * B[k * N + j];
            }
        }
    }

}
```

컴퓨터 시스템상, CPU -> L1 -> L2 -> L3 -> Memory 순서로 lookup 및 액세스를 하게 된다. 자주 접근하는 데이터에 대해 낮은 넘버의 캐시에서 캐시 미스가 자주 날수록 결과 속도는 늦어지게 된다. 이를 바탕으로, 행렬 크기를 조절함으로써 캐시 미스(capacity miss)를 일부러 유발할 수 있고, 거기서부터 캐시의 크기를 확인할 수 있다.

위의 코드는 B의 크기가 충분히 크다면 모든 값에서 캐시 미스가 발생한다. 이러한 코드에서 capacity miss가 발생한다면 성능 하락은 두드러질 것이다.

행렬의 크기를 주된 control로 했다. M,N,K의 값을 같게 하고, 그 숫자를 점점 키워서 성능

하락이 어디서 나타나는지를 확인했다. -n 10 옵션을 이용해서 평균을 구했고, 해당 시행을 10번 반복해서 그중 최대값을 결과값으로 제시했다.

만약 캐시가 private 캐시라면, 성능이 줄어드는 지점(즉 다음 캐시를 참조해야 하는 상황) 이전까지의 퍼포먼스가 스레드의 수에 비례해서 증가할 것이다. 그렇지 않다면, 성능이 줄어드는 지점 이전까지의 퍼포먼스가 스레드의 수에 비례해서 증가하지 않을 것이다.

결과는 다음과 같다. GFLOPS 자리에 있는 값은 각각 스레드를 1개, 4개, 16개로 증가시켜서 실험한 값이다. -O3 옵션을 두지 않고 측정한 결과 L3 캐시로 추정되는 부분에서 slurm이 종료되었기 때문에, -O3 조건을 병렬로 두어 확인했다.

M=N=K	size(KiB)	size(MiB)	speed(GLOPS)	speed(GFLOPS, -O3)
32	12	0.01	(0.3019 ,)	
36	15.19	0.01	(0.3174 ,)	
38	16.92	0.02	(0.3021 ,)	
40	18.75	0.02	(0.3013 ,)	
44	22.69	0.02	(0.3016 , 0.5598, 1.4762)	
48	27	0.03	(0.3213 , 0.6192, 1.9506)	
52	31.69	0.03	(0.3196 , 0.6785, 2.0534)	
56	36.75	0.04	(0.2640 , 0.4692, 1.9660)	
60	42.19	0.04	(0.2482 , 0.4888, 1.7580)	
64	48	0.05	(0.2522 , 0.5012, 0.9855)	
68	54.19	0.05	(0.2523 , 0.4722, 1.8391)	
72	60.75	0.06	(0.3199 , 0.5636, 1.7563)	
76	67.69	0.07	(0.3702 , 0.5576, 1.5677)	
128	192	0.19	(0.3499 , 1.0957, 2.2654)	(1.6371, 2.5517, 5.7148)
192	432	0.42	(0.4086 , 1.1439, 2.0124)	(1.1080, 3.9291, 10.3331)
256	768	0.75	(0.4151 , 1.4384, 3.1912)	(1.2989, 3.8217, 14.8296)
384	1728	1.69	(0.2996 , 1.4100, 3.0810)	(1.4946, 3.8320, 13.8151)
512	3072	3	(0.2730 , 0.9717, 3.2101)	(0.9489, 3.6061, 8.9439)
768	6912	6.75	(0.2045 , 0.6892, 2.2938)	(1.1992, 4.7690, 13.4004)
1024	12288	12	(0.2151 , 0.6324, 2.1745)	(0.6344, 2.5865, 8.6745)
1568	28812	28.14	(term. , 0.4895, 4.6130)	(0.8108, 3.0998, 10.8046)
2048	49152	48	(term. , 0.4889, 2.4895)	(term., 1.7026, 6.0956)

a) L1 cache 추정, private or shared?

첫 번째 붉은 색 부분이 L1 캐시의 최대 사이즈로 추정되는 부분으로, 약 36KiB에 해당한다. 이 이전까지의 성능을 보면 스레드가 4개일 때 대비 16개일 때 약 3배 이상 증가한 것을 확

인할 수 있다. (4배가 증가하지 않는 이유는, 스레드를 생성하고 합치는 등의 문제 때문인 것으로 추론할 수 있다.) 이는 L1 캐시가 private캐시라는 것을 방증한다.

b) L2 cache 추정, private or shared?

두 번째 붉은 색 부분이 L2 캐시로 추정되는 부분으로, 약 1.69MiB ~ 3MiB에 해당한다. 그 이전까지의 값 또한 L1 캐시일 때와 마찬가지로 3배 가량 증가하였으므로 private하다고 추측할 수 있다.

c) L3 cache 추정, private or shared?

세 번째 붉은 색 부분이 L3 캐시로 추정되는 부분으로, 약 48MiB 구간이다. 이 지점을 살펴보면 private 캐시라고 하기에는 성능 향상이 5배 가량 나타남을 볼 수 있다. 가설로는 확인할 수 없는 결과이지만, 이 부분이 L1, L2 부분에서 보이는 양상과 확연히 다르다는 점을 봤을 때 shared일 것으로 추론해본다. Private한 경우 B를 읽는 작업을 모든 스레드가 실행해야 하지만, shared인 경우에는 하나의 스레드만 시행하면 다른 스레드는 shared 캐시에서 데이터를 가져오면 되므로 오히려 더 효율적인 결과가 나오는 것이 아닐까 추론해 보았다.

d) 추정값과 실제값 비교하기

srun lscpu 및 검색을 통해 알아낸 캐시 크기는 다음과 같다. L1 캐시에서 약간의 차이가, L2 캐시에서 많은 차이가 발생했다. 이는 행렬 사이즈를 좀 더 촘촘히 하면 보완할 수 있는 부분으로 보인다.

L1d: 1MiB, private -> 32KiB per cpu

L1i: 32MiB, private -> 1MiB per cpu

L2: 48MiB, shared