과제 #3

홍경인

M1522.006700 확장형 고성능 컴퓨팅 (001)

October 29, 2024

1 Matrix Multiplication using OpenMP

1.1 병렬화 방식

전체 행렬곱 과정을 블록(block)으로 나누어, 이를 각 thread에 할당하는 방식으로 병렬화를 진행하였으며, 이 과정에서 블록 크기를 조절하여 캐시 사용을 최적화했다. OpenMP의 `#pragma omp parallel for` 지시어를 사용하여 가장 바깥쪽 루프의 반복을 병렬화하였으며, 이를 통해 각 thread가 계산을 독립적으로 수행하고, OpenMP가 이를 동기화하도록 처리하였다.

1.2 OpenMP의 스레드 생성 방식

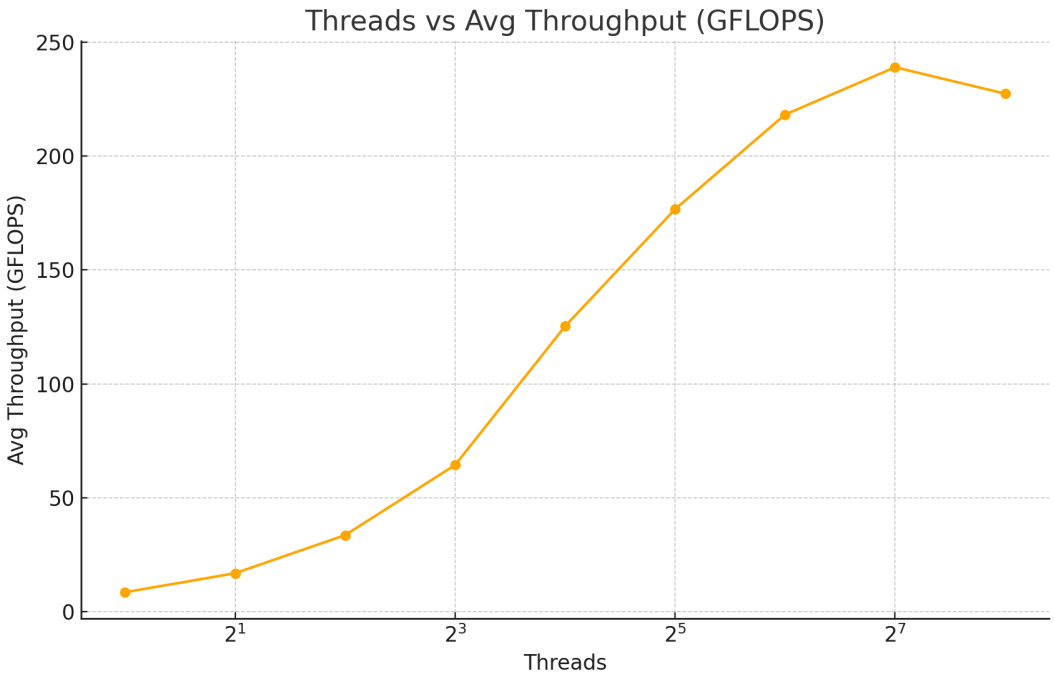
OpenMP는 프로그래머가 thread를 명시적으로 생성하지 않아도 thread를 생성하여 관리할 수 있다. 관련된 처리가 컴파일 타임과 런타임에 각각 다음과 같이 이루어진다:

- **컴파일러:** 컴파일러가 `#pragma omp`로 시작하는 OpenMP 지시어를 인식하면 OpenMP 런타임 시스템에서 인식할 수 있는 코드를 해당 지점에 삽입한다.
- **런타임 시스템:** 컴파일러가 삽입한 코드를 기반으로 해당 지점에서 여러 thread를 생성하여 운용한 후 병렬 처리가 끝나는 곳에서 합친다. 이처럼 전반적으로 하나의 master thread를 운용하되, 일부 지점에서 fork과 join이 일어난다.

1.3 Thread 수에 따른 성능 분석

Thread 개수를 1에서 256까지 변화시키며 측정한 결과는 다음과 같다. 각각 $M = N = K = 4096$ 환경에서 수행하였다.

No. of threads가 증가하는 초기에는 성능이 선형적으로 증가하는 것을 확인할 수 있다. 하지만 일정 개수를 초과하면 성능 향상이 둔화되거나 한계에 도달하는 것을 관찰할 수 있다. 이는



No. of threads	Avg. Throughput (GFLOPS)
1	8.561494
2	16.919459
4	33.630059
8	64.412747
16	125.335555
32	176.725116
64	218.261209
128	239.024639
256	227.430337

메모리 대역폭 한계로 메모리에서 데이터를 읽고 쓰는 지점에서 병목이 발생하거나, 작업을 분할하고 스레드를 관리하는 과정에서 오버헤드가 증가하기 때문인 것으로 판단된다.

1.4 OpenMP의 loop scheduling 방식

OpenMP에서 loop은 다양한 방식으로 각 thread에게 할당된다. 이와 관련하여 `static`, `dynamic`, `guided` 방식은 다음과 같이 작동한다:

- `static`: 각 thread에 task를 동일한 수만큼 할당.
- `dynamic`: 각 thread에 일정한 크기의 task를 동적으로 할당. Thread가 할당된 작업을

완료하면 새로운 작업을 요청하여 loop이 모두 처리될 때까지 진행.

- `guided`: `dynamic`과 유사하나 반복의 크기가 점차 줄어들어 병렬 처리가 진행됨에 따라 더 작은 단위로 동적 할당시킴.

각 scheduling별 성능 실험 결과는 아래 표와 같다. 실험은 32 threads, $M = N = K = 4096$으로 10회 진행하여 그 throughput의 평균을 GFLOPS로 나타낸 것이다.

Scheduling	Avg. throughput (GFLOPS)
Static	164.360643
Dynamic	186.888988
Guided	186.955060

2 Estimating Cache Size

Cache size를 확인하기 위해 제출 코드와는 별도로 아래와 같이 코드를 작성하여 실험을 진행하였다. 이때 cache miss가 더 빈번해져 `BLOCK_SIZE`에 따른 결과가 더 크게 차이나게 된다.

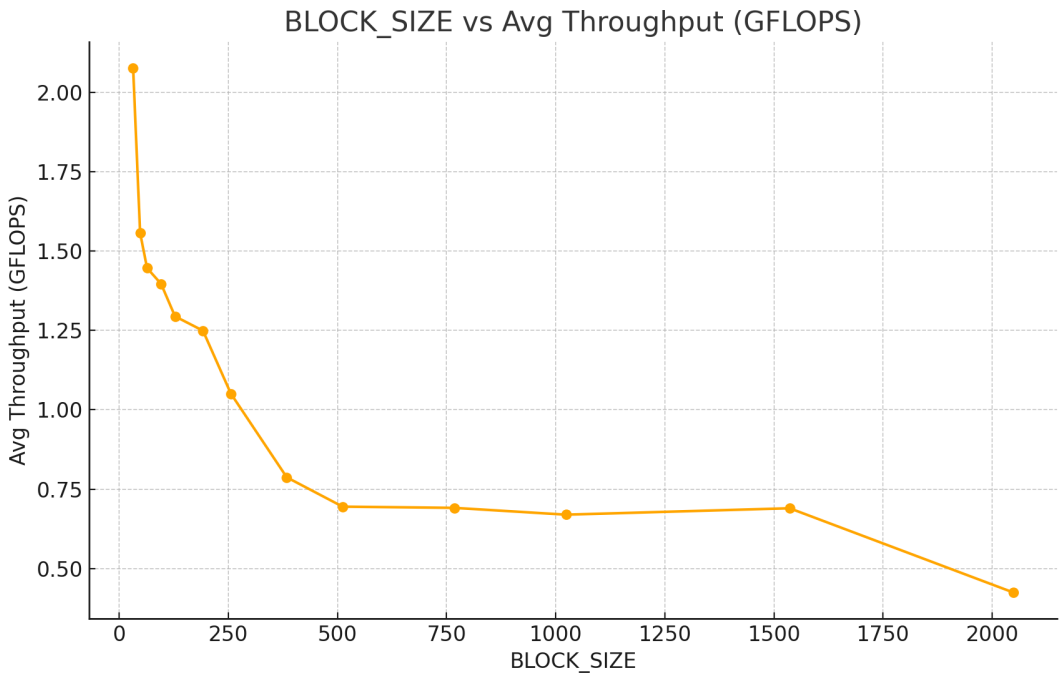
```
#define _GNU_SOURCE
#include "util.h"
#include <immintrin.h>
#include <omp.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define BLOCK_SIZE 32

void matmul(float *A, float *B, float *C, int M, int N, int K,
            int num_threads) {
    printf("\nBLOCK_SIZE: %d\n", BLOCK_SIZE);
    omp_set_num_threads(num_threads);
    #pragma omp parallel for schedule(guided) collapse(2)
    for (int ii = 0; ii < M; ii += BLOCK_SIZE) {
        for (int kk = 0; kk < K; kk += BLOCK_SIZE) {
            for (int jj = 0; jj < N; jj += BLOCK_SIZE) {
                int i_end = (ii + BLOCK_SIZE > M) ? M : ii + BLOCK_SIZE;
                int k_end = (kk + BLOCK_SIZE > K) ? K : kk + BLOCK_SIZE;
```

```
                int j_end = (jj + BLOCK_SIZE > N) ? N : jj + BLOCK_SIZE;
                for (int i = ii; i < i_end; ++i) {
                    for (int j = jj; j < j_end; ++j) {
                        float sum = C[i*N + j];
                        for (int k = kk; k < k_end; ++k) {
                            sum += A[i*K + k] * B[k*N + j];
                        }
                        C[i*N + j] = sum;
                    }
                }
            }
        }
    }
}
```

이를 여러 BLOCK_SIZE에 대하여 $M = N = K = 2048$로 수행하여 throughput을 구한 결과는 다음와 같다:



BLOCK_SIZE	Avg. Throughput (GFLOPS)
32	2.076139
48	1.556892
64	1.446096
96	1.396662
128	1.294191
192	1.249061
256	1.050633
384	0.787105
512	0.694735
768	0.690935
1024	0.669486
1536	0.689695
2048	0.424675

다음으로, cache의 private 여부를 확인하기 위해 제출 코드와는 별도로 아래와 같이 코드를 작성하여 실험을 진행하였다. 이때 여러 thread가 동일한 부분(`C[i*N + j]`)을 반복적으로 참조하게 되므로, 만약 특정 cache가 shared인 경우 coherence를 유지하기 위해 연산 자원이 많이 소모되게 된다.

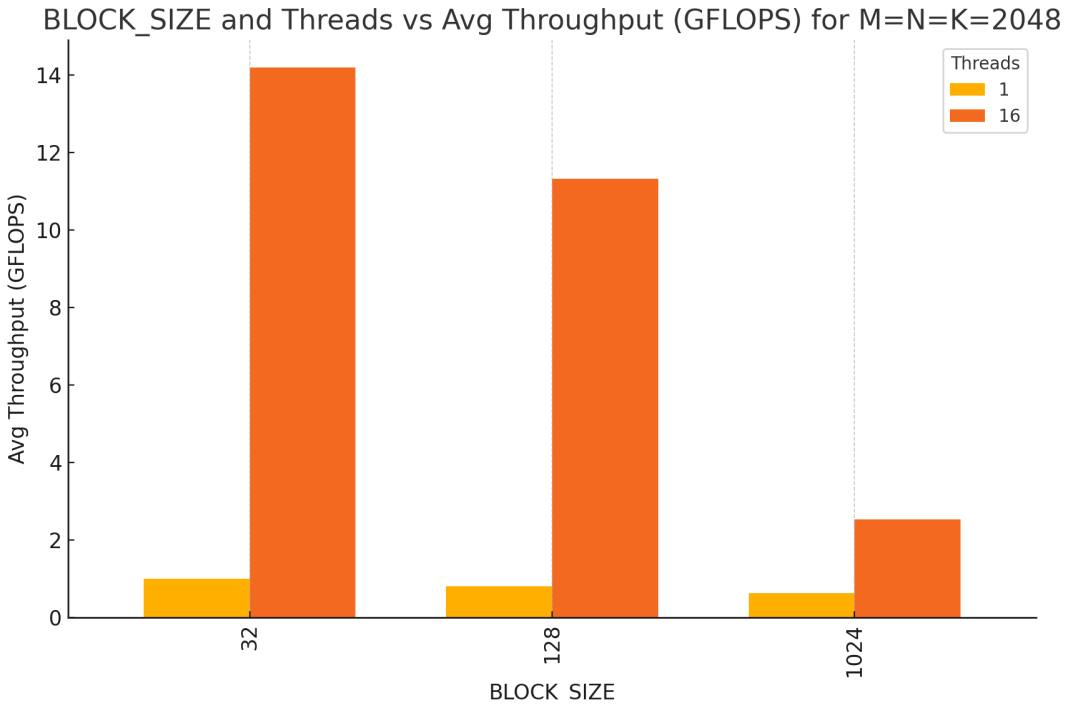
```
#define _GNU_SOURCE
#include "util.h"
#include <immintrin.h>
#include <omp.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define BLOCK_SIZE 32

void matmul(float *A, float *B, float *C, int M, int N, int K,
            int num_threads) {
    printf("\nBLOCK_SIZE: %d\n", BLOCK_SIZE);
    omp_set_num_threads(num_threads);
    #pragma omp parallel for schedule(guided) collapse(2)
    for (int ii = 0; ii < M; ii += BLOCK_SIZE) {
        for (int kk = 0; kk < K; kk += BLOCK_SIZE) {
            for (int jj = 0; jj < N; jj += BLOCK_SIZE) {
                int i_end = (ii + BLOCK_SIZE > M) ? M : ii + BLOCK_SIZE;
```

```
                int k_end = (kk + BLOCK_SIZE > K) ? K : kk + BLOCK_SIZE;
                int j_end = (jj + BLOCK_SIZE > N) ? N : jj + BLOCK_SIZE;
                for (int i = ii; i < i_end; ++i) {
                    for (int j = jj; j < j_end; ++j) {
                        for (int k = kk; k < k_end; ++k) {
                            C[i*N + j] += A[i*K + k] * B[k*N + j];
                        }
                    }
                }
            }
        }
    }
}
```

위 연산을 L1, L2, L3 캐시에서 수행하기 위해 `BLOCK_SIZE`를 32, 128, 1024로 각각 설정한 뒤, $M = N = K = 2048$에서 계산을 수행한 결과는 다음과 같다:



BLOCK_SIZE	No. of threads	Avg. Throughput (GFLOPS)
32	1	0.994141
32	16	14.191598
128	1	0.799232
128	16	11.318471
1024	1	0.635646
1024	16	2.537916

이를 다음과 같이 분석할 수 있다.

- **L1 cache**: BLOCK_SIZE가 32와 48 사이일 때 성능이 크게 하락하였으므로 L1 cache의 크기는 $3 \times 32^2 \times 4 = 12288$ byte와 $3 \times 48^2 \times 4 = 27648$ byte 사이일 것으로 추측할 수 있다. 이는 private cache로 추측되는데, L1 cache 안에서 연산이 이루어지도록 BLOCK_SIZE를 32로 두어 실험한 결과 thread 수가 physical core의 수와 같은 상황에서 throughput이 큰 폭으로 증가했기 때문이다.

- **L2 cache**: BLOCK_SIZE가 256와 384 사이일 때 성능이 크게 하락하였으므로 L2 cache의 크기는 $3 \times 256^2 \times 4 = 786432$ byte와 $3 \times 384^2 \times 4 = 1769472$ byte 사이일 것으로 추측할 수 있다. 이는 private cache로 추측되는데, L2 cache 안에서 연산이 이루어지도록 BLOCK_SIZE를 128로 두어 실험한 결과 thread 수가 physical core의 수와 같은 상황에서 throughput이 큰 폭으로 증가했기 때문이다. 만약 증가폭이 저조하다면 각 processor가 차지하는 cache 영역 간에 간섭이 일어나 multithreading의 효과가 크지 않을 것이다.

- **L3 cache**: BLOCK_SIZE가 1536와 2048 사이일 때 성능이 크게 하락하였으므로 L3 cache의 크기는 $3 \times 1536^2 \times 4 = 28311552$ byte와 $3 \times 2048^2 \times 4 = 50331648$ byte 사이일 것으로 추측할 수 있다. 이는 shared cache로 추측되는데, L3 cache 안에서 연산이 이루어지도록 BLOCK_SIZE를 1024로 두어 실험한 결과 thread 수가 physical core의 수와 같은 상황에서 throughput의 증가폭이 상대적으로 저조했기 때문이다. 이때 각 processor가 차지하는 cache 영역 간에 간섭이 일어나 multithreading의 효과가 크지 않았던 것으로 추측할 수 있다.

- **추정값과 실제값 비교**: 실제 L1, L2, L3 cache의 크기는 32768, 1048576, 23068672 byte이다. L1 cache를 축정하는 상황에서, 블록을 여러 번 나누어야 하므로 오버헤드가 발생하여 cache memory를 차지했을 가능성이 있다. 이때 L1 cache의 크기는 실제값보다 작게 추정된다. L2 및 L3 cache의 경우 적절히 추정된 것으로 분석된다.