EECE372 HW#6

20190445 허수범

May 30, 2024

1 Introduction

본 과제는 다양한 병렬 처리 방법을 통해 주어진 task를 수행하는 과제였습니다. OpenMP를 활용한 multithreading을 통해 vector addition과 dot product를 수행하는 것과 Neon을 활용한 SIMD를 통해 matrix multiplication을 수행하는 것이 본 과제의 목표였습니다.

2 Methods

2.1 Vector Addition

'vector_addition.c'의 'vec_slicing()' 함수는 Listing 1과 같이 구현하였습니다. for loop을 이용해 각 thread가 돌아가며 하나씩 sum을 처리하도록 하였습니다. 'thread_num'부터 시작해 'num_threads'만큼 index를 건너뛰며 'ARRAY_SIZE'전까지 sum을 수행하도록 구현하였고, 결과들이 서로 independent하므로 critical region을 설정할 필요가 없었습니다.

```
void vec_slicing(double *x, double *y, double *z) {
    omp_set_num_threads(6);
    // Write Your Code
#pragma omp parallel
    {
        int thread_num = omp_get_thread_num();
        int num_threads = omp_get_num_threads();
        for(int i = thread_num; i < ARRAY_SIZE; i += num_threads){// perform slicing
            z[i] = x[i] + y[i];
        }
    }
}
```

Listing 1: 'vec_slicing()'.

'vec_chunking()' 함수는 Listing 2과 같이 구현하였습니다. 총 thread의 갯수만큼 전체 array를 나눈 chunk별로 각 thread가 연산을 수행하도록 하였습니다. 이 과정에서 각 chunk의 시작 index와 끝 index를 계산해주는 과정이 필요했습니다. 각 chunk의 index들을 계산한 후에는 for loop을 통해 addition을 수행하였습니다. 또한, 배열 바깥의 메모리에 접근하지 않도록 end 인덱스를 설정할 때 if문을 사용해 예외처리를 해주었습니다.

```c
void vec_chunking(double *x, double *y, double *z) {
    omp_set_num_threads(6);
    // Write Your Code
#pragma omp parallel
    {
        int thread_num = omp_get_thread_num();
        int num_threads = omp_get_num_threads();
        int chunk_size = ARRAY_SIZE / num_threads;
        int start = thread_num * chunk_size;
        int end;
        if(thread_num = num_threads - 1){
            end = ARRAY_SIZE;
        } else{
            end = start + chunk_size;
        }
        for(int i = start; i < end; i++){// perform chunking
            z[i] = x[i] + y[i];
        }
    }
}
```

Listing 2: 'vec_chunking()'.

2.2 Dot Product

'dot_product.c'의 'dot_omp()' 함수는 Lsiting 3과 같이 구현하였습니다. Dot product를 구하는 task의 경우, 위의 vector addition과 다르게 각 thread들의 연산 결과가 독립적이지 않습니다. 그렇기 때문에, data race가 발생할 수 있는데, 이를 방지하기 위해 dependency가 존재하는 부분에 '#pragma omp critical' directive를 활용해 critical region을 설정해주었습니다. For loop에서는 각 thread가 local sum을 구하도록 하였고, 이 local sum을 모두 합칠 때 thread간 dependency가 발생하기 때문에 global sum을 구하는 부분에는 위에서 언급한 '#pragma omp critical' directive를 사용하여 구현하였습니다. Local sum은 각 thread가 담당하는 부분의 결과입니다. 각 thread는 배열의 일부 원소의 연산만을 담당하기에, 해당 부분 원소들의 곱의 합만을 미리 연산하고, 이를 이후 critical region에서 global sum에 accumulate하는 방식을 사용하였습니다.

```c
double dotp_omp(double *x, double *y) {
    omp_set_num_threads(6);

    double global_sum = 0.0;

#pragma omp parallel
    {
        int num_thread = omp_get_num_threads();
        int thread_ID = omp_get_thread_num();

        // Write Your Code
        double local_sum = 0.0; // set local sum

        for(int i = thread_ID; i < ARRAY_SIZE; i += num_thread){// slicing
            local_sum += x[i] * y[i]; // get local sum
        }

#pragma omp critical
        {
            global_sum += local_sum;
        }
    }

    return global_sum;
}
```

Listing 3: 'dot_omp()'.

2.3 Matrix Multiplication with NEON

'Neon'의 'main.c'에서 Neon으로 matrix multiplication을 수행하는 부분의 일부는 Listings 4, 5의 snippet들에서 확인할 수 있습니다. 먼저 Listing 4은 배열에서 각 row를 추출하는 부분입니다. Listing 4에서는 arr1의 row를 추출하는 snippet만이 적혀 있지만, 두 행렬의 row들을 모두 'int64x8_t' 벡터에 추출해 주었습니다. 이후에는 Listing 5에서 볼 수 있듯이, 'int64x8_t' type의 'sum' 벡터를 선언하여 부분적인 sum을 구할 벡터를 설정하였고, 이 'sum'에 arr1의 한 row의 각 원소들과 그에 대응하는 arr2의 row들을 곱하고 accumulate하였습니다. arr1의 한 row에 대한 accumulation을 마친 후, sum을 'ans_neon'의 알맞은 위치에 저장해 주었습니다. 이를 arr1의 각 row에 대해 수행하는 방식으로 matrix multiplication을 구현하였습니다.

```
// get all the rows of arr1
int16x8_t arr1_0 = vld1q_s16(arr1);
int16x8_t arr1_1 = vld1q_s16(arr1 + 8);
int16x8_t arr1_2 = vld1q_s16(arr1 + 16);
int16x8_t arr1_3 = vld1q_s16(arr1 + 24);
int16x8_t arr1_4 = vld1q_s16(arr1 + 32);
int16x8_t arr1_5 = vld1q_s16(arr1 + 40);
int16x8_t arr1_6 = vld1q_s16(arr1 + 48);
int16x8_t arr1_7 = vld1q_s16(arr1 + 56);
```

Listing 4: Snippet from 'main.c'(getting rows).

```
// set accumulator
int16x8_t sum = vmovq_n_s16(0);

// perform multiplication and accumulation for elements in row 1 of arr1 and all the
    rows of arr2
sum = vmlaq_lane_s16(sum, arr2_0, vget_low_s16(arr1_0), 0);
sum = vmlaq_lane_s16(sum, arr2_1, vget_low_s16(arr1_0), 1);
sum = vmlaq_lane_s16(sum, arr2_2, vget_low_s16(arr1_0), 2);
sum = vmlaq_lane_s16(sum, arr2_3, vget_low_s16(arr1_0), 3);
sum = vmlaq_lane_s16(sum, arr2_4, vget_high_s16(arr1_0), 0);
sum = vmlaq_lane_s16(sum, arr2_5, vget_high_s16(arr1_0), 1);
sum = vmlaq_lane_s16(sum, arr2_6, vget_high_s16(arr1_0), 2);
sum = vmlaq_lane_s16(sum, arr2_7, vget_high_s16(arr1_0), 3);
vst1q_s16(ans_neon, sum);
```

Listing 5: Snippet from 'main.c'(calculation).

3 Results

코드 실행 결과들은 Figures 1, 2, 3과 같습니다. 모두 정상적으로 프로그램이 수행되며, 연산이 정상적으로 수행되었다는 표시인 'PASS'가 터미널에 display되는 것을 확인할 수 있습니다.

4 Conclusion & Discussion

본 과제의 결론과 Discussion은 각 task별로 서술하겠습니다.

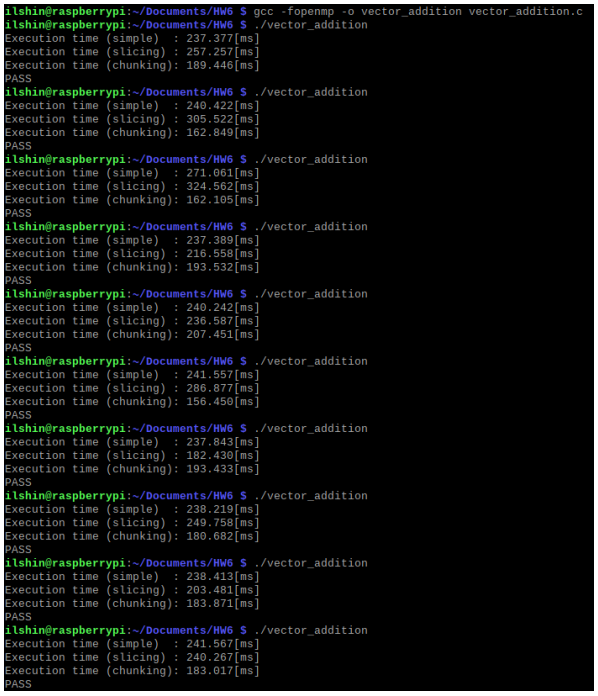

Figure 1: Execution results for vector addition.

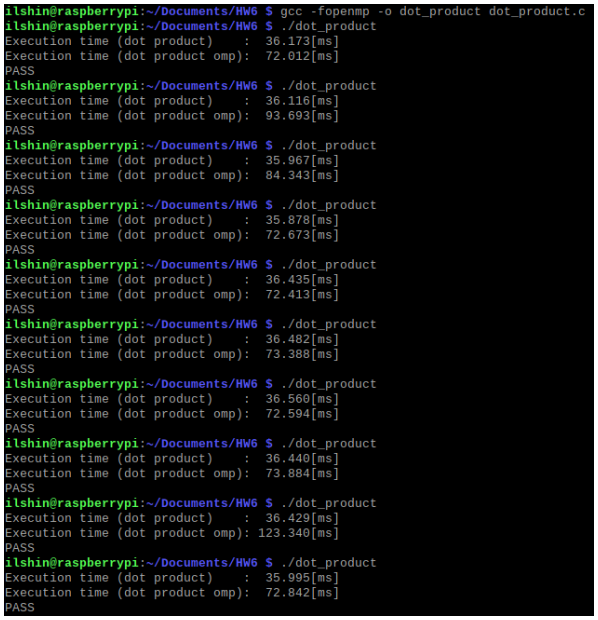

Figure 2: Execution results for dot product.

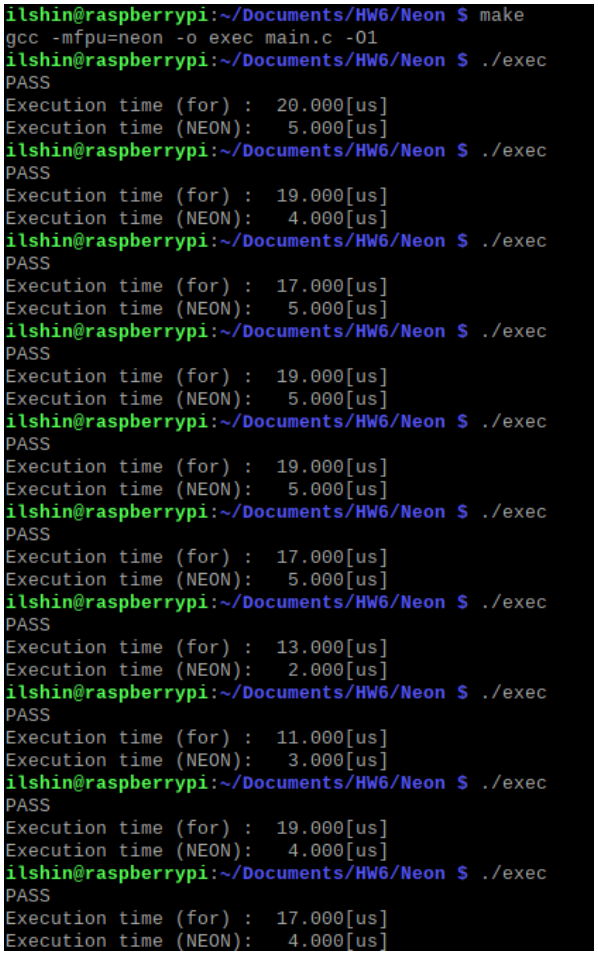

Figure 3: Execution results for neon matrix multiplication.

4.1 Vector Addition

위의 Figure 1의 결과들을 평균 내어 보면, simple의 결과는 평균 242.409ms, slicing의 결과는 평균 250.3299ms, chunking의 결과는 181.2433ms입니다. simple과 slicing의 결과는 큰 차이가 없고, chunking의 결과는 훨씬 빠른 것을 확인할 수 있습니다. Slicing의 경우, 순차적으로 for loop이 돌며 thread들이 연산을 하기 때문에, 각 thread가 정해진 분량 만큼 연산을 수행하는 chunking에 비해 느리고, 큰 배열에 대해 비효율적이었던 것으로 보입니다.

4.2 Dot Product

위의 Figure 1의 결과들을 평균 내어 보면, simple의 결과는 평균 36.2475ms, OpenMP의 결과는 평균 81.1182ms입니다. 예상치 못한 결과로, simple이 OpenMP를 활용한 알고리즘보다 현저히 낮은 수행 시간을 가지는 것을 확인할 수 있습니다. 이는 critical section의 문제로 보여 추가 실험을 진행하였습니다.

원인을 찾아보기 위해 thread의 갯수를 수정하며 다시 실험을 진행한 결과, thread의 갯수가 4개인 경우 OpenMP의 결과는 평균 60.7102ms였고, 2개인 경우 OpenMP의 결과는 평균 39.521ms였습니다. Thread의 갯수가 줄어들수록 실행 시간이 줄어드는 경향이 뚜렷하며, 이는 '#pragma omp critical'의 영향으로 thread의 갯수가 늘어날수록 비효율적인 연산을 한다는 것의 증거로 볼 수 있습니다. Thread를 1로 설정한 경우, simple의 수행 시간과 매우 비슷한 결과가 나오지만, 이는 병렬화를 수행한다는 본 과제의 목표와 맞지 않은 것으로 보고 결과를 첨부하지 않았습니다.

결국 data race를 해결하는 과정에서 순차적인 연산 수행이 필요하기에 위와 같은 결과가 나온 것을 예상됩니다.

4.3 Matrix Multiplication with NEON

위의 Figure 1의 결과들을 평균 내어 보면, simple의 결과는 평균 17.1us, NEON의 결과는 평균 4.2us입니다. NEON의 SIMD를 활용한 결과가 약 4배가량 빠른 것으로 보아, 많은 병렬 연산이 필요한 경우 SIMD의 위력이 강력하다는 것을 확인할 수 있었습니다. OpenMP를 통한 Instruction 병렬화에 비해 data의 병렬화를 이용하는 연산이 구현하기에는 더욱 어려움이 있지만, 강력한 성능을 낼 수 있다는 것을 확인하였습니다.