목차

1. 실행 결과

Lab 5-2. Matrix Multiplication with Shared Memory

```
C:₩Program Files₩PowerShell
PS H:\Dev\koreatech-assignment\MulticoreProgramming\Assignment5\Lab5-2\cmake-build-release\Release> ./L
ab5-2.exe 1024 256 2048 32
[row, k, col, blockSize] = [1024, 256, 2048, 32]

*        DS_timer Report        *
* The number of timer = 6, counter = 6
**** Timer report ****
Total : 12.24550 ms (12.24550 ms)
Kernel : 5.85710 ms (5.85710 ms)
Data Transfer Time (Host > Device) : 4.92190 ms (4.92190 ms)
Data Transfer Time (Device > Host) : 1.46180 ms (1.46180 ms)
Timer host : 3030.58360 ms (3030.58360 ms)
Timer omp : 175.32310 ms (175.32310 ms)
**** Counter report ****
*        End of the report      *
CUDA & OpenMP works well!
```

```
#define TO_INDEX(row, col, width) ((row) * (width) + (col))
#define GET(pointer, row, col, width) (pointer[TO_INDEX(row, col, width)])
#define GET_B(pointer, row, col) GET(pointer, row, col, blockSize)
```

```
__global__ void matrixMul(const float *a, const float *b, float *c, int row, int k, int col, int blockSize) {
    unsigned int rowIx = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int colIx = blockDim.y * blockIdx.y + threadIdx.y;

    extern __shared__ float sharedTotal[];
    float *sharedA = sharedTotal;
    float *sharedB = sharedA + blockSize * blockSize;

    unsigned int threadRow = threadIdx.x;
    unsigned int threadCol = threadIdx.y;

    float result = 0;
    for (int block = 0; block < ceil(k / (float) blockSize); block++) {
        int offset = block * blockSize;

        GET_B(sharedA, threadRow, threadCol) = (rowIx < row && offset + threadCol < k) ? a[TO_INDEX(rowIx, offset + threadCol, k)] : 0;
        GET_B(sharedB, threadRow, threadCol) = (colIx < col && offset + threadRow < k) ? b[TO_INDEX(offset + threadRow, colIx, col)] : 0;
        __syncthreads();

        for (int i = 0; i < blockSize; i++) {
            result += __fmul_rn(GET(sharedA, threadRow, i, blockSize), GET(sharedB, i, threadCol, blockSize));
        }
        __syncthreads();
    }

    if (!(rowIx < row && colIx < col)) {
        return;
    }

    c[TO_INDEX(rowIx, colIx, col)] = result;
}
```

기존 5-1 과제의 구조를 이용하여 기본적인 데이터 초기화 및 GPU 데이터 전송을 구현하였고, 공유 메모리의 용량 한계(64KB)를 극복하기 위해, sharedA(a 행렬), sharedB(b 행렬)로 구성된 2개의 공유 메모리 배열에 a와 b를 블록 단위(blockSize * blockSize)로 부분 복사한 뒤 이 블록 단위의 행렬곱 연산이 k 길이 (A 행렬의 열, B 행렬의 행 길이)에 대해서 (k / blockSize)번 (블록 단위 만큼) 반복되게 구성하였다.

row	k	col	blockSi	Total	Kernel	Data Transfer Time (Host > Device)	Data Transfer Time (Device > Host)	Host	Host om	Type
256	2048	4096	32	29.4758	20.4558	8.0043	1.0074	103356.1	3605.431	Lab5-1
256	2048	4096	32	32.1173	23.1743	8.1039	0.831	103030	3885.901	Lab5-2
256	1024	4096	32	15.8977	10.3527	4.6788	0.8626	43698.64	433.5801	Lab5-1
256	1024	4096	32	17.9188	12.1775	4.911	0.8224	43533.17	469.7673	Lab5-2
256	512	4096	32	8.7562	5.3939	2.5984	0.7588	17974.8	187.8576	Lab5-1
256	512	4096	32	9.5578	5.9791	2.734	0.8403	18581.37	173.0368	Lab5-2
256	256	4096	32	5.5639	2.8118	1.9301	0.8188	8732.731	111.7868	Lab5-1
256	256	4096	32	6.1327	3.5809	1.7715	0.7768	8747.344	93.3843	Lab5-2
256	4096	2048	32	28.8775	20.4639	7.9167	0.4881	13588.24	3443.75	Lab5-1
256	4096	2048	32	31.9195	23.217	8.1764	0.5182	14193.96	3662.917	Lab5-2
256	2048	2048	32	15.2991	10.3746	4.4097	0.5076	7093.237	442.1235	Lab5-1
256	2048	2048	32	16.7947	12.1297	4.2185	0.4417	7428.544	412.1588	Lab5-2
256	1024	2048	32	8.579	5.4777	2.6243	0.4736	3158.472	154.6305	Lab5-1
256	1024	2048	32	8.9084	6.0068	2.4598	0.4359	3157.746	169.6851	Lab5-2
256	512	2048	32	4.9311	2.8616	1.5602	0.5063	1612.297	91.3724	Lab5-1
256	512	2048	32	5.3815	3.2456	1.5462	0.586	1622.162	91.5875	Lab5-2
256	256	2048	32	3.1026	1.6978	0.9757	0.4264	765.4569	61.0218	Lab5-1
256	256	2048	32	3.1325	1.7801	0.904	0.4458	758.5406	56.5821	Lab5-2
256	4096	1024	32	15.249	10.2707	4.7114	0.2625	7067.49	498.5021	Lab5-1
256	4096	1024	32	16.4558	11.6448	4.5406	0.2639	6973.47	522.9428	Lab5-2
256	2048	1024	32	8.45	5.3678	2.6529	0.4243	3247.301	188.7902	Lab5-1
256	2048	1024	32	8.897	5.9425	2.684	0.2665	3210.622	169.6985	Lab5-2
256	1024	1024	32	4.4457	2.7593	1.4242	0.2588	1584.874	91.5315	Lab5-1
256	1024	1024	32	4.7805	3.1023	1.4125	0.2615	1601.47	93.4885	Lab5-2
256	512	1024	32	3.0058	1.512	0.9979	0.4929	777.3542	43.6358	Lab5-1
256	512	1024	32	2.7856	1.6718	0.8541	0.2576	787.4216	61.5551	Lab5-2

[행렬 크기를 변경해가며 테스트한 프로그램 실행 결과 (RTX 3080, Ryzen 9 3900X)]

row, k, col: 행렬의 크기

matrixSize: row * col

Total: Kernel + Data Transfer Time + Host 시간

Kernel: 커널 부분 (행렬 곱) 부분만의 실행 시간

Data Transfer Time: 데이터 이동간 걸리는 시간

Host: 호스트의 행렬 곱 소요 시간

Host omp: OpenMP를 이용하여 계산한 경우 소요 시간

*. 시간 단위는 밀리 초를 사용

CPU 직렬로 구성한 코드 대비는 3000배 가량, CPU 병렬로 구성한 코드보다는 100배가량 성능 향상이 있었지만, 다양한 행렬 크기의 케이스에 대해서 Lab5-1보다 성능이 동일하거나 더 근소하게 낮은 결과를 보였다. (Lab5-1 대비 0.99x, 098x) 다양한 원인이 있을 수 있겠지만, 공유 메모리에 데이터를 옮기는 오버헤드에 비해서, 공유 메모리 접근으로 얻을 수 있는 성능 향상이 크지 않았던 것으로 추정된다.

2. 소감

처음으로 성능 개선이 잘 이루어지지 않았던 과제였다. 여러가지 시도를 해봤었는데, 5-1 과제에 비하여 성능 향상이 이루어지지 않았다. 비록 큰 성능 개선은 없었지만, 이전 강의에서 데이터 접근 빈도를 올리기 위해 행렬 곱의 연산 정의를 더 복잡한 수식으로 바

꾸었던 것처럼 행렬 곱보다 메모리 접근이 더 자주 이루어지는 연산을 수행하면 이런 구조에서도 성능 향상이 있지 않을지와 공유 메모리를 어떻게 어떤 구조로 배치하면, 더 효율적으로 메모리를 활용할 수 있을지에 대한 것처럼 최적화에 대한 다양한 고민을 해볼 수 있는 과제였던 것 같다.