목차

1. 구현 방법 및 실행 결과

Lab. 7-1 Trapezoidal Rule on GPU

```
__global__ void trapWithAtomic(double a, double b, unsigned int n, double delta, double *sum) {
    unsigned int tid = blockDim.x * blockIdx.x + threadIdx.x;
    atomicAdd(sum, 1 <= tid && tid < n ? f(a + tid * delta) : 0.0);
}
```

```
timer.onTimer(TIMER_CUDA_ATOMIC);
trapWithAtomic<<<(unsigned int) ceil(n / (double) BLOCK_SIZE), BLOCK_SIZE>>>(a, b, n, delta, dSum);
cudaDeviceSynchronize();
cudaMemcpy(&deviceSum, dSum, sizeof(double), cudaMemcpyDeviceToHost);
deviceSum = (deviceSum + (f(a) + f(b)) / 2.0) * delta;
timer.offTimer(TIMER_CUDA_ATOMIC);
printf("Atomic Sum: %lf\n", deviceSum);
```

명령줄 인자로 받은 a, b, n를 이용하여 구간 합을 구하는 커널 로직을 구성하였다. 각 스레드가 한 개의 구간을 맡아 atomic 연산을 이용하여 합을 구하도록 하였으며, 스레드 레이아웃은 1차원 그리드와 1차원 스레드를 사용하여, 최대 블록 크기를 넘는 구간합을 계산할 수 있도록 구성하였다.

Lab 7-2. Optimized Trap. Rule on GPU

```
__global__ void trapWithReduction(double a, double b, unsigned int n, double delta, double *sum) {
    unsigned int tid = blockDim.x * blockIdx.x + threadIdx.x;

    __shared__ double blockSums[BLOCK_SIZE];
    blockSums[threadIdx.x] = 1 <= tid && tid < n ? f(a + tid * delta) : 0.0;
    __syncthreads();

    for (int offset = BLOCK_SIZE / 2; offset > 0; offset /= 2) {
        if (threadIdx.x < offset)
            blockSums[threadIdx.x] += blockSums[threadIdx.x + offset];
        __syncthreads();
    }

    if (!threadIdx.x)
        atomicAdd(sum, blockSums[0]);
}
```

```
cudaMemset(dSum, 0, sizeof(double));
timer.onTimer(TIMER_CUDA_REDUCTION);
trapWithReduction<<<(unsigned int) ceil(n / (double) BLOCK_SIZE), BLOCK_SIZE>>>(a, b, n, delta, dSum);
cudaDeviceSynchronize();
cudaMemcpy(&deviceSum, dSum, sizeof(double), cudaMemcpyDeviceToHost);
deviceSum = (deviceSum + (f(a) + f(b)) / 2.0) * delta;
timer.offTimer(TIMER_CUDA_REDUCTION);
printf("Reduction Sum: %lf\n", deviceSum);
```

Lab 7-1 코드에서는 커널에서 실행되는 모든 스레드가 단일한 합 변수를 두고 atomic 합 연산을 수행하기 때문에, 병목이 발생한다. 따라서 블록 내 스레드끼리 Reduction Sum을 이용하여 블록당 합을 계산하고, 이를 블록의 대표 스레드 한 개가 블록 합을 atomic하게 합하도록 하여, 전역 합 변수에 대한 경합이 전체 스레드가 아니라 블록별 스레드 1개에

대해서 이루어지도록 변경하였다.

실행 결과

```
f(x) = x * x
range = (-10.000000, 10.000000), n = 4294967295
Serial Sum: 666.666667
Atomic Sum: 666.666667
Reduction Sum: 666.666667

*         DS_timer Report         *
* The number of timer = 3, counter = 3
**** Timer report ****
Timer Serial : 3180.36410 ms (3180.36410 ms)
Timer CUDA (Atomic) : 7063.73180 ms (7063.73180 ms)
Timer CUDA (Reduction) : 168.57930 ms (168.57930 ms)
**** Counter report ****
*         End of the report       *
```

(BlockSize: 64, CPU: Ryzen 9 3900X, GPU: RTX 3080에서 측정됨)

Atomic (Lab. 7-1) 버전은 CPU 보다 2.2배 가량 느려진 결과를 보였고, Reduction (Lab. 7-2) 버전은 18.9배 성능 향상이 된 것을 확인할 수 있었다. Atomic 버전의 경우 원자 연산을 활용했는데도, 다루는 스레드가 많아 병목이 발생한 것으로 추정되고, 이를 개선하여 경합을 줄인 Reduction 버전은 예상대로 큰 성능 향상이 있었다. 블록 크기 측면에서는 2의 배수별로 성능을 측정하였는데 블록 크기가 64일 때 가장 좋은 성능을 보여주었다. (2 ~ 32, 128 ~ 1024 크기에서는 성능이 감소함)

2. 소감

이번 과제에서는 OpenMP에서 다루었던 것처럼, GPU에서의 병목과 경합 상황을 해결하는 방법에 대해서 배울 수 있었다. 지난 번 과제에서 다루었던 Reduction 기법을 다시 복습할 수 있어서 좋았다.