

# 기초 GPU 프로그래밍 HW1 보고서

20211584 장준영

## 1. 사용된 GPU

NVIDIA GeForce RTX 3070 Ti 8GB

## 2. 구현 및 작동 방식

시간 측정에 있어 공평하기 위해, 모든 방법의 실질적인 계산 부분만 함수 안에 담았다. 기본적으로 (1) 평균을 위한 배열 값의 총 합, (2) 분산을 위한 배열 값의 제곱의 총 합, (3) 최댓값, (4) 최솟값을 계산한다.

```
float *original_A = new float[N];
prepare_input_data(original_A, N);

float *avgsum = new float[N];
float *varsum = new float[N];
float *maxi = new float[N];
float *mini = new float[N];
```

original\_A에는 원본 배열 값이 담기고, 밑의 네 개의 배열에는 앞서 이야기 한 각각의 계산 값을 담을 예정이다. 이후 평균은 avgsum의 대표값을 N으로 나누어서, 분산은 varsum의 대표값을 N으로 나눈 것에 평균의 제곱을 빼서 구할 예정이다.

[1] HW1\_host

```
void HW1_host(int n,
              float *avgsum, float *varsum, float *maxi, float *mini,
              float *average, float *variance, float *maximum, float *minimum)
{
    for (int i = 0; i < n; i++)
        varsum[i] *= varsum[i];

    float _avgsum = 0.0,
        _varsum = 0.0,
        _maximum = -1.0,
        _minimum = 2.0;

    for (int i = 0; i < n; i++)
    {
        /* AVERAGE, VARIANCE */
        _avgsum += avgsum[i];
        _varsum += varsum[i];
        /* MAXIMUM */
        _maximum = (_maximum < maxi[i]) ? maxi[i] : _maximum;
        /* MINIMUM */
        _minimum = (_minimum > mini[i]) ? mini[i] : _minimum;
    }

    *average = (_avgsum / n);
    *variance = (_varsum / n) - ((*average) * (*average));
    *maximum = _maximum;
    *minimum = _minimum;

    return;
}
```

host에서의 계산은 직관적이다. `_avgsum`에는 총합, `_varsum`에는 제곱의 총합, `_maximum`에는 최댓값, `_minimum`에는 최솟값을 저장하고, 포인터로 넘어온 네 개의 요소에 각각의 계산 방법으로 값을 계산해 할당해주면 된다.

## [2] HW1\_reduce1

```
float *d_avgsum, *d_varsum, *d_maxi, *d_mini, *d_temp;
cudaMalloc((void **)&d_avgsum, N * sizeof(float));
cudaMalloc((void **)&d_varsum, N * sizeof(float));
cudaMalloc((void **)&d_maxi, N * sizeof(float));
cudaMalloc((void **)&d_mini, N * sizeof(float));
cudaMalloc((void **)&d_temp, N * sizeof(float));
```

```
/* DEVICE */
init_arr(avgsum, varsum, maxi, mini, original_A, N);
cudaMemcpy(d_avgsum, avgsum, N * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_varsum, varsum, N * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_maxi, maxi, N * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_mini, mini, N * sizeof(float), cudaMemcpyHostToDevice);
```

우선 device에서 사용할 배열 공간을 `cudaMalloc`과 `cudaMemcpy`를 통해 할당해야 한다. `cudaMalloc`을 통해 GPU 메모리 공간을 할당하고, `cudaMemcpy`를 통해 CPU의 데이터를 GPU로 전송할 수 있다.

```
__global__ void HW1_reduce1(int n,
                           float *avgsum, float *varsum, float *maxi, float *mini,
                           bool isFirst /* for avoiding double squaring in varsum */)
{
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    stride = gridDim.x * blockDim.x;
    float _avgsum = 0.0,
          _varsum = 0.0,
          _maximum = -1.0,
          _minimum = 2.0;

    for (int i = tid; i < n; i += stride)
    {
        /* AVERAGE, VARIANCE */
        _avgsum += avgsum[i];
        if (isFirst)
            varsum[i] *= varsum[i];
        _varsum += varsum[i];
        /* MAXIMUM */
        _maximum = (_maximum < maxi[i]) ? maxi[i] : _maximum;
        /* MINIMUM */
        _minimum = (_minimum > mini[i]) ? mini[i] : _minimum;
    }

    avgsum[tid] = _avgsum;
    varsum[tid] = _varsum;
    maxi[tid] = _maximum;
    mini[tid] = _minimum;

    return;
}
```

`__global__`로 선언된 `HW1_reduce1`은 예제 코드인 `reduce1`과 동일한 방식으로 작동한다. 전체 값을 구간 내에 모으기 위해, `stride` 씩 건너 뛰며 대표값을 찾아 맨 앞(tid)에 저장한다. 이 과정을 구간의 크기가 1일 때까지 반복하면 하나의 대표값을 구할 수 있다. 추가된 구현점은 파라미터인 `bool isFirst`인데, `varsum`에 제곱의 합을 구하기 위해 `_varsum`

`+= (varsum[i] * varsum[i])` 라인을 넣고 여러 번의 커널 콜을 할 경우, 그 단계마다 값을 제공하게 되어 잘못된 값이 나온다. 따라서, 최초의 커널 콜인 경우에만 `varsum[i]`의 값을 제공하고 이후에는 평범하게 합을 진행하였다.

```
HW1_reduce1<<<blocks, threads>>>(N, d_avgsum, d_varsum, d_maxi, d_mini, true);
HW1_reduce1<<<1, threads>>>(blocks * threads, d_avgsum, d_varsum, d_maxi, d_mini, false);
HW1_reduce1<<<1, 1>>>(threads, d_avgsum, d_varsum, d_maxi, d_mini, false);
```

예제 코드의 `reduce1`과 마찬가지로 세 번 호출해 대표값을 구하였다.

### [3] HW1\_thrust

```
void HW1_thrust(int n,
               thrust::device_vector<float> t_A,
               float *average, float *variance, float *maximum, float *minimum)
{
    float _avgsum = 0.0,
          _varsum = 0.0,
          _maximum = -1.0,
          _minimum = 2.0;

    _avgsum = thrust::reduce(t_A.begin(), t_A.end(), (float)0.0, thrust::plus<float>());
    _varsum = thrust::transform_reduce(t_A.begin(), t_A.end(), square_float(), (float)0.0, thrust::plus<float>());
    _maximum = thrust::reduce(t_A.begin(), t_A.end(), (float)-1.0, thrust::maximum<float>());
    _minimum = thrust::reduce(t_A.begin(), t_A.end(), (float)2.0, thrust::minimum<float>());

    *average = (_avgsum / n);
    *variance = (_varsum / n) - ((*average) * (*average));
    *maximum = _maximum;
    *minimum = _minimum;

    return;
}

struct square_float
{
    __host__ __device__
    float operator()(const float &x) const { return x * x; }
};
```

HW1\_thrust에서는 CUDA thrust 라이브러리를 이용하기 때문에, 구현이 매우 단순하다. `thrust::reduce`와 `thrust::transform_reduce`를 사용하였다.

## 3. 계산 결과 및 분석

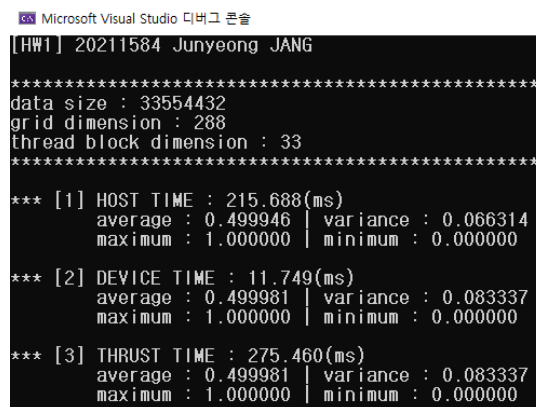
[1]  $N = 1 \ll 25$ , blocks = 288, threads = 256 (default)

```
Microsoft Visual Studio 디버그 콘솔
[HW1] 20211584 Junyeong JANG
*****
data size : 33554432
grid dimension : 288
thread block dimension : 256
*****
*** [1] HOST TIME : 214.973(ms)
      average : 0.499960 | variance : 0.066290
      maximum : 1.000000 | minimum : 0.000000
*** [2] DEVICE TIME : 2.505(ms)
      average : 0.500007 | variance : 0.083327
      maximum : 1.000000 | minimum : 0.000000
*** [3] THRUST TIME : 230.873(ms)
      average : 0.500007 | variance : 0.083327
      maximum : 1.000000 | minimum : 0.000000
```

`reduce1` 예제 코드에 기본으로 설정 되어있는 값이다. 우선 host에서 구한 계산값과 device/thrust에서 구한 계산값이 서로 조금 다른 것을 확인할 수 있다. 이는 부동소수점 연산이 불가피하게 오차를 발생시키기 때문이다. 또한 순차적으로 모든 값을 더한 host

에서의 계산 방식과 병렬적으로 stride 만큼 건너뛰며 값을 더한 device에서의 계산 방식이 서로 다르기 때문에, 더해지는 순서가 달라지면서 값에 오차가 발생할 수 있다. 복잡한 수학 연산의 경우, GPU와 CPU는 정밀도나 함수 구현 등에서 차이가 있기 때문에 오차가 발생하기도 한다. 후술할 thread block의 크기를 다르게 한 실험의 경우에도, device와 thrust의 계산 값은 동일하기 때문에 이 두 방법의 계산 순서가 동일하다고 추측할 수 있다. 연산 속도는 device, host, thrust 순으로 느려진다. GPU의 병렬 처리를 가장 효율적으로 구현한 device에서의 연산이 다른 방법보다 대략 열 배 정도 빠르므로, SIMT 형태의 연산을 할 때는 GPU의 병렬성을 활용하는 것이 좋다.

[2]  $N = 1 \ll 25$ , blocks = 288, threads = 33

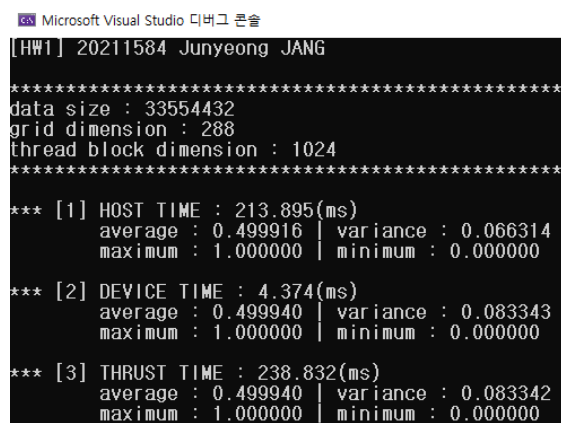


```

Microsoft Visual Studio 디버그 콘솔
[HW1] 20211584 Junyeong JANG
*****
data size : 33554432
grid dimension : 288
thread block dimension : 33
*****
*** [1] HOST TIME : 215.688(ms)
      average : 0.499946 | variance : 0.066314
      maximum : 1.000000 | minimum : 0.000000
*** [2] DEVICE TIME : 11.749(ms)
      average : 0.499981 | variance : 0.083337
      maximum : 1.000000 | minimum : 0.000000
*** [3] THRUST TIME : 275.460(ms)
      average : 0.499981 | variance : 0.083337
      maximum : 1.000000 | minimum : 0.000000
  
```

thread block의 크기를 33로 설정하였다. warp 사이즈인 32보다 1 큰 숫자로, 교수님께서 효율적이지 않다고 말씀하신 '32로 나누어 떨어지지 않는 수' 중 최대한 작은 값으로 설정해보았다. 앞서 2.5ms 정도로 처리되던 연산이 11.7ms까지 걸리는 것을 확인할 수 있다. 32로 나누어 떨어지지 않는 크기이기 때문에 마지막 warp가 완전히 채워지지 않은 채 단 한 thread만 할당되어 사용되었고, 크기 자체가 너무 작아 GPU의 모든 SM을 충분히 활용하지 못하는 등의 이유로 속도 저하가 나타난 것으로 보인다.

[3]  $N = 1 \ll 25$ , blocks = 288, threads = 1024



```

Microsoft Visual Studio 디버그 콘솔
[HW1] 20211584 Junyeong JANG
*****
data size : 33554432
grid dimension : 288
thread block dimension : 1024
*****
*** [1] HOST TIME : 213.895(ms)
      average : 0.499916 | variance : 0.066314
      maximum : 1.000000 | minimum : 0.000000
*** [2] DEVICE TIME : 4.374(ms)
      average : 0.499940 | variance : 0.083343
      maximum : 1.000000 | minimum : 0.000000
*** [3] THRUST TIME : 238.832(ms)
      average : 0.499940 | variance : 0.083342
      maximum : 1.000000 | minimum : 0.000000
  
```

thread block의 크기를 최대값인 1024로 설정하였다. 이론상 단일 블록에 대해 최대한 많은 thread를 활용할 수 있고, GPU occupancy도 높아질 것으로 보인다. 하지만, 기본값으로 설정했을 때의 2.5ms보다 느린 4.4ms가 나왔다. 이는 GPU 메모리 접근의 병목 현상이나 레지스터 사용량의 과도한 증가 등으로 인한 성능 저하가 영향을 끼친 것으로 추측된다.