기초 GPU 프로그래밍 HW3 보고서

20211584 장준영

1. 환경

CPU : AMD Ryzen 5 5600X 6-Core Processor 3.70GHz

RAM : DDR4 32GB

GPU : NVIDIA GeForce RTX 3070 Ti 8GB

IDE : Visual Studio 2022 CUDA C/C++

2. 결과 확인 방법

```
/* Preprocessing declarations - Please read item 2. in the report carefully. **********************/
/* 1. */
#define VIDEO_LENGTH 10
#define ENABLE_GF_CPU
/* 2. */
#define IMAGESET_LENGTH 25
#define NUM_STREAMS_M 16
/* Image file */
#define WRITE_FILE
const string INPUT_FILE_LOC = "Data\\";
const string INPUT_FILE_NAME = "Image_0_7360_4832";
const string INPUT_FILE_EXT = ".jpg";
#define INPUT_IMG ((INPUT_FILE_LOC + INPUT_FILE_NAME + INPUT_FILE_EXT).c_str())
const string OUTPUT_FILE_LOC = "C:\\usr\\S20211584\\";
const string OUTPUT_FILE_EXT = ".png";
#define OUTPUT_IMG(num) ((OUTPUT_FILE_LOC + INPUT_FILE_NAME + "_" + num + OUTPUT_FILE_EXT).c_str())
/*****************************************************************************************************/
```

코드 상단에 결과 확인을 돕는 전처리 선언부가 존재한다. 각각 다음과 같은 의미를 갖는다.

- VIDEO_LENGTH : 1번 문제의 '영상 길이'를 의미한다. 1번 문제의 커널(GF_*)은 영상 처리와 유사한 동작을 위해 해당 횟수만큼 호출된다.

- ENABLE_GF_CPU : main에서 구현한 모든 함수를 순차적으로 호출하는데, 알 수 없는 이유로 GF_CPU 호출 시에 바로 다음에 호출되는 GF_SHARED_1의 수행 시간이 길어졌다. GF_SHARED_1의 올바른 수행 시간을 확인하기 위해, 해당 정의가 존재할 때만 GF_CPU를 호출하도록 작성하였다.

(※ 만약 채점 중 GF_SHARED_* 커널의 수행 시간이 비정상적인-보고서에 작성된 것과 비율에 있어 크게 차이 나는- 경우, 'GF_CPU의 수행 시간을 확인할 때와 결과 이미지 파일을 생성할 때'는 ENABLE_GF_CPU를 정의해주시고, 그렇지 않은 경우 정의를 주석 처리 해주세요.)

- IMAGESET_LENGTH : 2번 문제의 '영상 길이'를 의미한다. 해당 길이 만큼의 IO_Images 배열을 생성하여 영상처럼 사용한다.

- NUM_STREAMS_M : 2번 문제에서 사용할 non-default stream의 개수이다.

- WRITE_FILE : 결과 이미지 파일을 작성할지 결정한다. 해당 정의가 있을 때만 결과 이미지 파일을 C:₩usr₩S20211584 디렉토리에 저장한다.

(※ WRITE_FILE 역시 IMAGESET_NONDEFAULT_M의 수행 시간에 영향을 주는 것으로 추측되니, 수행 시간이 비정상적인 경우 정의를 주석 처리 해주세요. GF_CPU의 수행 시간 및 결과 이미지를 확인할 때를 제외하고는 두 개의 정의 모두 제거하는 것을 추천 드립니다.)

3. 구현 방법

```
__constant__ float filter_weight[25];
#define IDX(x, y, nx, ny) (nx * (y < 0 ? 0 : (y ≥ ny ? ny - 1 : y)) + (x < 0 ? 0 : (x ≥ nx ? nx - 1 : x)))
```

가우시안 필터 가중치를 상수 값으로 디바이스에서 이용할 수 있는 filter_weight, 이미지의 2차원 인덱스를 이미지 범위 내의 1차원 row-major 인덱스로 변환해주는 IDX 매크로이다.

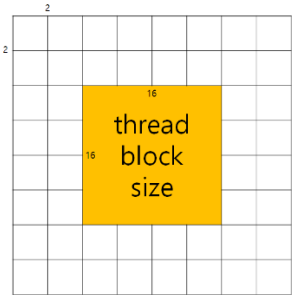
[1-1] GF_CPU

```
void GF_CPU(const uchar4 *__restrict input_image, uchar4 *__restrict output_image, int nx, int ny)
{
    int y_offset[5], x_offset[5];
    for (int y = 0; y < ny; y++)
    {
        for (int x = 0; x < nx; x++)
        {
            y_offset[0] = max(0, y - 2), y_offset[1] = max(0, y - 1), y_offset[2] = y, y_offset[3] = min(ny - 1, y + 1), y_offset[4] = min(ny - 1, y + 2);
            x_offset[0] = max(0, x - 2), x_offset[1] = max(0, x - 1), x_offset[2] = x, x_offset[3] = min(nx - 1, x + 1), x_offset[4] = min(nx - 1, x + 2);
            float4 v = make_float4(0.0f, 0.0f, 0.0f, 0.0f);
            for (int i = 0; i < 5; i++)
            {
                for (int j = 0; j < 5; j++)
                {
                    uchar4 pixel_in = input_image[nx * y_offset[i] + x_offset[j]];
                    float weight = filter_5[GAUSSIAN_FILTER_5][i * 5 + j];
                    v.x += pixel_in.x * weight;
                    v.y += pixel_in.y * weight;
                    v.z += pixel_in.z * weight;
                    v.w += pixel_in.w * weight;
                }
            }
            output_image[IDX(x, y, nx, ny)] = {(unsigned char)min(255, max(0, (unsigned int)(v.x + 0.5f))),
                                               (unsigned char)min(255, max(0, (unsigned int)(v.y + 0.5f))),
                                               (unsigned char)min(255, max(0, (unsigned int)(v.z + 0.5f))),
                                               (unsigned char)min(255, max(0, (unsigned int)(v.w + 0.5f)))};
        }
    }
}
```

GF_CPU의 구현은 GF_GLOBAL과 완전히 동일하지만, CPU 프로세서의 특성 상 단일 프로세스가 이미지 배열을 모두 탐색하면서 필터링 작업을 수행한다는 차이가 있다. 이미지 내의 모든 픽셀에 대해, 필터 반지름 내의 모든 픽셀 값(offset)을 참조하여 가중치만큼 곱하고 더해준다. 해당 픽셀에 대한 필터링 작업을 마쳤다면, 출력 이미지에 바뀐 값을 쓴다.

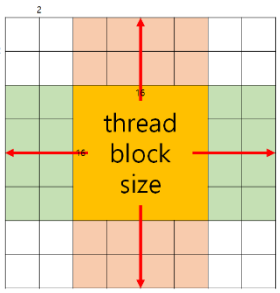
[1-2] GF_SHARED_1

```cpp
__global__ void
GF_SHARED_1(const uchar4 *__restrict input_image, uchar4 *__restrict output_image, int nx, int ny)
{
    __shared__ uchar4 tile[16 + 4][16 + 4];

    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int bx = blockIdx.x * blockDim.x;
    int by = blockIdx.y * blockDim.y;

    int x = bx + tx, y = by + ty;
    if (x < 0 || y < 0 || x ≥ nx || y ≥ ny)
        return;

    tile[ty + 2][tx + 2] = input_image[IDX(x, y, nx, ny)];
```



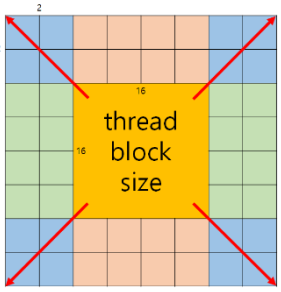
우선 shared memory 영역에 블록 사이즈(16*16)와 상하좌우 필터 반지름(2+2)을 모두 포함하는 크기의 2차원 배열을 선언한다. 이후 쓰레드 블록 내의 해당 쓰레드의 위치를 tx, ty에, 쓰레드 블록의 최좌측 최상단(오프셋)을 bx, by에, 전체 이미지에 대한 해당 쓰레드의 위치를 x, y에 저장한다. 가장 먼저, 타일의 중심부에 모든 쓰레드가 자신의 위치에 해당하는 픽셀을 저장한다.

```cpp
    if (tx < 2)
        tile[ty + 2][tx] = input_image[IDX(x - 2, y, nx, ny)];
    else if (tx ≥ 16 - 2)
        tile[ty + 2][tx + 4] = input_image[IDX(x + 2, y, nx, ny)];

    if (ty < 2)
        tile[ty][tx + 2] = input_image[IDX(x, y - 2, nx, ny)];
    else if (ty ≥ 16 - 2)
        tile[ty + 4][tx + 2] = input_image[IDX(x, y + 2, nx, ny)];
```



다음으론 타일의 외곽 모서리를 채운다. 먼저 쓰레드 블록의 양 옆 모서리에 존재하는 쓰레드 네 줄을 시켜 근접한 외곽 모서리(초록색)를 채우도록 한다. 이후 같은 방법으로 쓰레드 블록의 위 아래 모서리에 존재하는 쓰레드 네 줄을 시켜 근접한 외곽 모서리(주황색)를 채우도록 한다.

```cpp
    if (tx < 2 && ty < 2)
        tile[ty][tx] = input_image[IDX(x - 2, y - 2, nx, ny)];
    else if (tx < 2 && ty ≥ 16 - 2)
        tile[ty + 4][tx] = input_image[IDX(x - 2, y + 2, nx, ny)];
    else if (tx ≥ 16 - 2 && ty < 2)
        tile[ty][tx + 4] = input_image[IDX(x + 2, y - 2, nx, ny)];
    else if (tx ≥ 16 - 2 && ty ≥ 16 - 2)
        tile[ty + 4][tx + 4] = input_image[IDX(x + 2, y + 2, nx, ny)];

    __syncthreads();
```

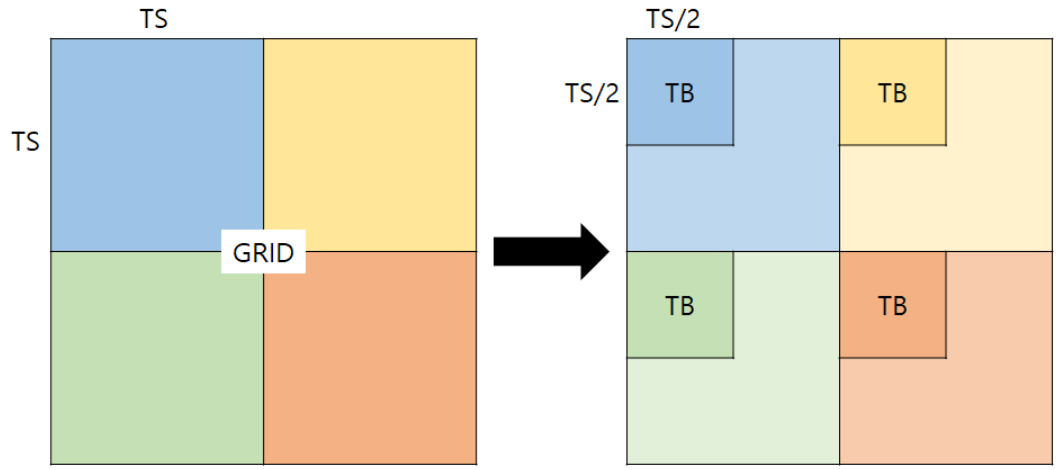


마지막으로 타일의 네 꼭짓점을 채운다. 이 역시 쓰레드 블록의 네 꼭짓점에 존재하는 쓰레드 16개를 시켜 근접한 꼭짓점(파란색)을 채우도록 한다. 이 과정을 거치면, 쓰레드 블록의 모든 쓰레드가 자신 위치의 픽셀에 필터링을 하기 위해 필요한 모든 픽셀을 shared memory 타일에 저장할 수 있다. 모든 타일을 채우고 난 후엔 __syncthreads를 통해 쓰레드 블록을 동기화한다.

```
    float4 v = make_float4(0.0f, 0.0f, 0.0f, 0.0f);
    int c_index = 0;
    for (int dy = -2; dy <= 2; dy++)
    {
        for (int dx = -2; dx <= 2; dx++)
        {
            uchar4 pixel_in = tile[ty + 2 + dy][tx + 2 + dx];
            float weight = filter_weight[c_index++];
            v.x += weight * pixel_in.x;
            v.y += weight * pixel_in.y;
            v.z += weight * pixel_in.z;
            v.w += weight * pixel_in.w;
        }
    }

    output_image[IDX(x, y, nx, ny)] = make_uchar4((unsigned char)min(255, max(0, int(v.x + 0.5f))),
                                                  (unsigned char)min(255, max(0, int(v.y + 0.5f))),
                                                  (unsigned char)min(255, max(0, int(v.z + 0.5f))),
                                                  (unsigned char)min(255, max(0, int(v.w + 0.5f))));
}
```

이후 타일을 이용해, 쓰레드마다 필요한 픽셀을 불러와 가중치만큼 곱하고 더한다. 필터링이 적용된 픽셀은 출력 이미지에 저장한다.

[1-3] GF_SHARED_N



GF_SHARED_1과 유사하지만, 쓰레드 당 네 개의 픽셀을 처리한다. 타일에 정보를 불러오는 것도, 필터링 이후 출력 이미지에 적용하는 것도 한 쓰레드가 인접한 네 개의 픽셀을 담당하게 한다. 그리드와 쓰레드 블록의 크기 관계는 위 이미지와 같다.

```
__global__ void
GF_SHARED_N(const uchar4 *__restrict__ input_image, uchar4 *__restrict__ output_image, int nx, int ny)
{
    __shared__ uchar4 tile[16 + 4][16 + 4];

    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int bx = blockIdx.x * (blockDim.x * 2);
    int by = blockIdx.y * (blockDim.y * 2);

    int x = bx + 2 * tx;
    int y = by + 2 * ty;

    if (x < 0 || y < 0 || x >= nx || y >= ny)
        return;
```

GF_SHARED_1과 동일하게 shared memory 영역에 tile을 선언하고, tx, ty, bx, by, x, y로 위치를 저장한다. 여기서 달라진 점은 블록 오프셋을 찾기 위해 블록 크기에 2를 곱하는 점과, 처리할 위치인 x, y를 특정할 때 tx, ty에 2를 곱하여 더하는 것이다. 이는 위 이미지에서 설명했듯이 하나의 쓰레드가 인접한 네 개의 픽셀을 처리하기 때문으로, 절반으로 줄어든 블록 크기에 맞게 블록 크기의 두 배 만큼 뛰어야 다음 블록이 나오고, 다음 쓰레드의 위치가 한 쓰레드의 처리 범위인 두 칸 뒤이다.

```
#pragma unroll 2
    for (int dy = 0; dy < 2; dy++)
    {
#pragma unroll 2
        for (int dx = 0; dx < 2; dx++)
        {
            int lx = tx * 2 + dx;
            int ly = ty * 2 + dy;
            tile[ly + 2][lx + 2] = input_image[IDX(x + dx, y + dy, nx, ny)];

            if (lx < 2)
            {
                tile[ly + 2][lx] = input_image[IDX(x + dx - 2, y + dy, nx, ny)];
            }
            else if (lx ≥ 16 - 2)
            {
                tile[ly + 2][lx + 4] = input_image[IDX(x + dx + 2, y + dy, nx, ny)];
            }

            if (ly < 2)
            {
                tile[ly][lx + 2] = input_image[IDX(x + dx, y + dy - 2, nx, ny)];
                if (lx < 2)
                    tile[ly][lx] = input_image[IDX(x + dx - 2, y + dy - 2, nx, ny)];
                if (lx ≥ 16 - 2)
                    tile[ly][lx + 4] = input_image[IDX(x + dx + 2, y + dy - 2, nx, ny)];
            }
            else if (ly ≥ 16 - 2)
            {
                tile[ly + 4][lx + 2] = input_image[IDX(x + dx, y + dy + 2, nx, ny)];
                if (lx < 2)
                    tile[ly + 4][lx] = input_image[IDX(x + dx - 2, y + dy + 2, nx, ny)];
                if (lx ≥ 16 - 2)
                    tile[ly + 4][lx + 4] = input_image[IDX(x + dx + 2, y + dy + 2, nx, ny)];
            }
        }
    }
    __syncthreads();
```

이후 x, y에 대해 처리할 네 픽셀의 위치 lx, ly를 설정한다. 그 다음의 과정은 GF_SHARED_1의 타일을 채우는 과정과 동일하다. 단순 네 번의 반복이기 때문에 #pragma unroll을 통해 모두 언롤링 하였다. 모든 타일을 채우고 난 후엔 __syncthreads를 통해 쓰레드 블록을 동기화한다.

```
#pragma unroll 2
    for (int dy = 0; dy < 2; dy++)
    {
#pragma unroll 2
        for (int dx = 0; dx < 2; dx++)
        {
            int lx = tx * 2 + dx;
            int ly = ty * 2 + dy;

            float4 v = make_float4(0.0f, 0.0f, 0.0f, 0.0f);
            int c_index = 0;
            for (int dyf = -2; dyf <= 2; dyf++)
            {
                for (int dxf = -2; dxf <= 2; dxf++)
                {
                    uchar4 pixel_in = tile[ly + 2 + dyf][lx + 2 + dxf];
                    float weight = filter_weight[c_index++];
                    v.x += weight * pixel_in.x;
                    v.y += weight * pixel_in.y;
                    v.z += weight * pixel_in.z;
                    v.w += weight * pixel_in.w;
                }
            }

            output_image[IDX(x + dx, y + dy, nx, ny)] = make_uchar4((unsigned char)min(255, max(0, int(v.x + 0.5f))),
                                                                    (unsigned char)min(255, max(0, int(v.y + 0.5f))),
                                                                    (unsigned char)min(255, max(0, int(v.z + 0.5f))),
                                                                    (unsigned char)min(255, max(0, int(v.w + 0.5f))));
        }
    }
}
```

타일을 모두 채우고 나선, 역시 하나의 쓰레드가 네 개의 픽셀에 대해 필터링을 하고 출력 이미지에 쓴다.

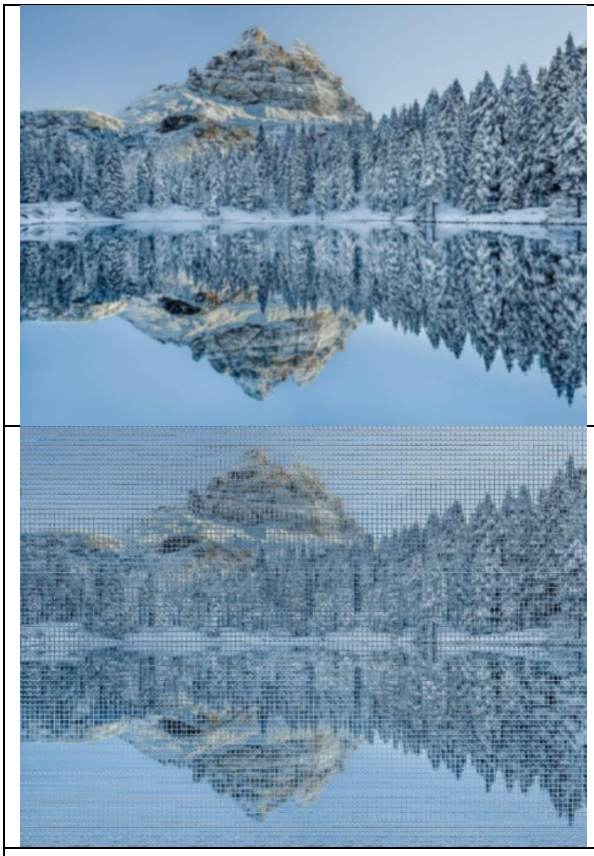
[1-4] GF_COMPARE

```
__global__ void
GF_COMPARE(const uchar4 *__restrict__ input1, const uchar4 *__restrict__ input2, uchar4 *__restrict__ output, int nx, int ny)
{
    auto idx = [&nx](int y, int x)
    { return y * nx + x; };

    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x < 0 || y < 0 || x >= nx || y >= ny)
        return;

    float4 out = make_float4(0.0f, 0.0f, 0.0f, 0.0f);
    uchar4 in1 = input1[idx(y, x)];
    uchar4 in2 = input2[idx(y, x)];
    out.x = abs(in1.x - in2.x);
    out.y = abs(in1.y - in2.y);
    out.z = abs(in1.z - in2.z);
    output[idx(y, x)] = {(unsigned char)min(255, max(0, (unsigned int)(out.x + 0.5f))),
                         (unsigned char)min(255, max(0, (unsigned int)(out.y + 0.5f))),
                         (unsigned char)min(255, max(0, (unsigned int)(out.z + 0.5f))),
                         (unsigned char)255};
}
```

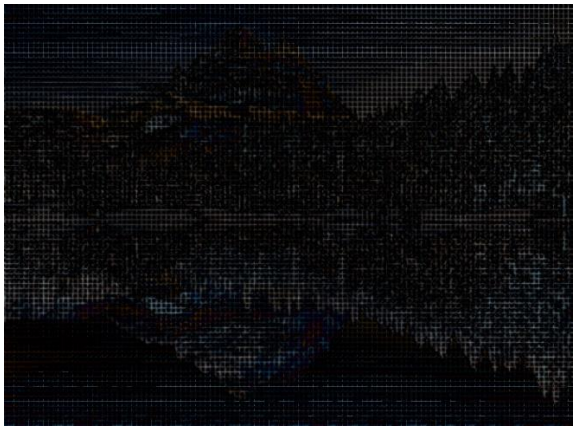
두 이미지의 차이를 비교하는 함수 역시 커널에서 수행하도록 작성하였다. 이미지의 모든 픽셀에 대해, R, G, B 값의 차이를 계산하여 출력 이미지에 작성한다.

```
threadx = 16;
thready = 16;
threads = {threadx, thready, 1};
blocks = {(io_images_shared.width + threads.x - 1) / threads.x,
          (io_images_shared.height + threads.y - 1) / threads.y, 1};

unsigned char *input1, *input2, *output;
cudaMalloc((void **)&input1, io_images_cmp1.data_bytes);
cudaMalloc((void **)&input2, io_images_cmp1.data_bytes);
cudaMalloc((void **)&output, io_images_cmp1.data_bytes);

cudaMemcpy(input1, io_images_global.output.data, io_images_global.data_bytes, cudaMemcpyHostToDevice);
cudaMemcpy(input2, io_images_shared.output.data, io_images_shared.data_bytes, cudaMemcpyHostToDevice);
GF_COMPARE<<<blocks, threads>>>((uchar4 *)input1, (uchar4 *)input2, (uchar4 *)output, io_images_cmp1.width, io_images_cmp1.height);
cudaMemcpy(io_images_cmp1.output.data, output, io_images_cmp1.data_bytes, cudaMemcpyDeviceToHost);
```

예시로 위의 과정을 거쳐, input1과 input2에 비교 대상 이미지를 넣으면 output에 비교 결과 이미지가 저장된다.



비교 대상



비교 결과

[2-1] IMAGESET_DEFAULT

```
void IMAGESET_DEFAULT(IO_Images *video, int video_length)
{
    uchar4 *d_input, *d_output;
    cudaMalloc(&d_input, video[0].data_bytes);
    cudaMalloc(&d_output, video[0].data_bytes);
    dim3 threads = {16, 16, 1};
    dim3 blocks = {(video[0].width + threads.x - 1) / threads.x,
                   (video[0].height + threads.y - 1) / threads.y, 1};

    CHECK_TIME_START(_start, _freq);
    for (int i = 0; i < video_length; i++)
    {
        /* H2D */
        cudaMemcpy(d_input, video[i].input.data, video[i].data_bytes, cudaMemcpyHostToDevice);
        /* KERNEL */
        GF_GLOBAL<<<blocks, threads>>>((uchar4 *)d_input, (uchar4 *)d_output, video[i].width, video[i].height);
        /* D2H */
        cudaMemcpy(video[i].output.data, d_output, video[i].data_bytes, cudaMemcpyDeviceToHost);
    }
    CHECK_TIME_END(_start, _end, _freq, _compute_time);

    cudaFree(d_input);
    cudaFree(d_output);
}
```

디바이스에 영상의 한 프레임(이미지 한 장)을 저장할 수 있는 메모리 d_input, d_output을 할당한다. 이후 영상의 시작부터 끝까지 모든 이미지에 대해, H2D->KERNEL->D2H 연산을 default stream에서 synchronous 하게 수행한다. 마지막으로 할당한 메모리를 해제한다.

[2-2] IMAGESET_NONDEFAULT_M

```
void IMAGESET_NONDEFAULT_M(IO_Images *video, int video_length)
{
    uchar4 *d_input[IMAGESET_LENGTH];
    uchar4 *d_output[IMAGESET_LENGTH];
    uchar4 *h_input[IMAGESET_LENGTH];
    uchar4 *h_output[IMAGESET_LENGTH];
    /* Pinned mem (host) */
    for (int i = 0; i < IMAGESET_LENGTH; i++)
    {
        cudaMallocHost(&h_input[i], video[i].data_bytes);
        cudaMallocHost(&h_output[i], video[i].data_bytes);
    }
    for (int i = 0; i < video_length; i++)
        cudaMemcpy(h_input[i], video[i].input.data, video[i].data_bytes, cudaMemcpyHostToHost);
    /* I/O mem (device) */
    for (int i = 0; i < IMAGESET_LENGTH; i++)
    {
        cudaMalloc(&d_input[i], video[i].data_bytes);
        cudaMalloc(&d_output[i], video[i].data_bytes);
    }
    /* Streams */
    cudaStream_t streams[NUM_STREAMS_M];
    for (int i = 0; i < NUM_STREAMS_M; i++)
        cudaStreamCreate(&streams[i]);
```

Asynchronous 한 명령 수행을 위해, 호스트 메모리를 pinned memory 영역에 잡는다. 또, 독립된 non-default stream을 NUM_STREAMS_M개 생성한다.

```
CHECK_TIME_START(_start, _freq);
for (int i = 0; i < video_length; i++)
{
    int stream_idx = i % NUM_STREAMS_M;
    /* H2D */
    cudaMemcpyAsync(d_input[i], h_input[i], video[i].data_bytes, cudaMemcpyHostToDevice, streams[stream_idx]);
    /* KERNEL */
    GF_GLOBAL<<<blocks, threads, 0, streams[stream_idx]>>>((uchar4 *)d_input[i], (uchar4 *)d_output[i], video[i].width, video[i].height);
    /* D2H */
    cudaMemcpyAsync(h_output[i], d_output[i], video[i].data_bytes, cudaMemcpyDeviceToHost, streams[stream_idx]);
}
cudaDeviceSynchronize();
CHECK_TIME_END(_start, _end, _freq, _compute_time);
```

이미지 한 장 마다 스트림 번호를 바꿔가면서, H2D->KERNEL->D2H 연산을 asynchronous 하게 수행한다.

```
    for (int i = 0; i < NUM_STREAMS_M; i++)
        cudaStreamSynchronize(streams[i]);
    for (int i = 0; i < NUM_STREAMS_M; i++)
        cudaStreamDestroy(streams[i]);
    for (int i = 0; i < video_length; i++)
        cudaMemcpy(video[i].output.data, h_output[i], video[i].data_bytes, cudaMemcpyHostToHost);
    for (int i = 0; i < IMAGESET_LENGTH; i++)
    {
        cudaFree(d_input[i]);
        cudaFree(d_output[i]);
        cudaFreeHost(h_input[i]);
        cudaFreeHost(h_output[i]);
    }
}
```

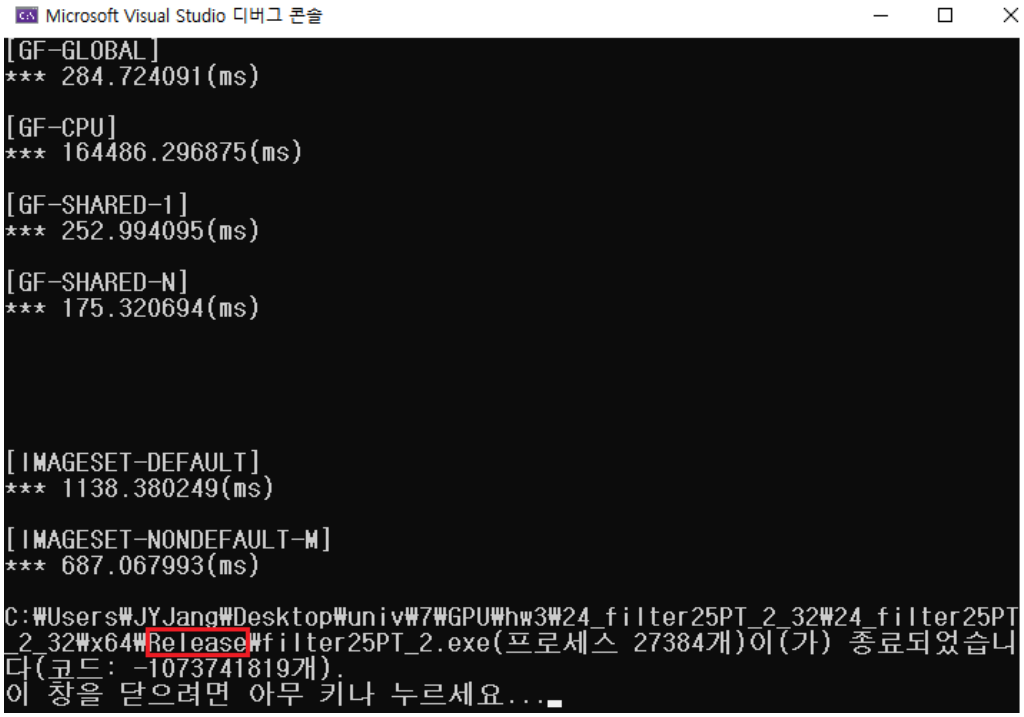
연산이 모두 마무리되면 스트림을 제거하고 pinned memory 영역에 저장된 결과를 output으로 복사한다. 이후 할당한 메모리 영역을 모두 해제한다.

4. 결과 및 분석

기준이 되는 실험 환경은 다음과 같다.

```
/* Preprocessing declarations - Please read item 2. in the report carefully. **********************/
/* 1. */
#define VIDEO_LENGTH 100
#define ENABLE_GF_CPU
/* 2. */
#define IMAGESET_LENGTH 25
#define NUM_STREAMS_M 16
/* Image file */
#define WRITE_FILE
const string INPUT_FILE_LOC = "Data\\";
const string INPUT_FILE_NAME = "Image_0_7360_4832";
const string INPUT_FILE_EXT = ".jpg";
#define INPUT_IMG ((INPUT_FILE_LOC + INPUT_FILE_NAME + INPUT_FILE_EXT).c_str())
const string OUTPUT_FILE_LOC = "C:\\usr\\S20211584\\";
const string OUTPUT_FILE_EXT = ".png";
#define OUTPUT_IMG(num) ((OUTPUT_FILE_LOC + INPUT_FILE_NAME + "_" + num + OUTPUT_FILE_EXT).c_str())
/*************************************************************************************************/
```

이 환경에서 프로그램을 20번 실행한 후, 결과를 평균내었다.



실행 결과 이미지

GF-GLOBAL	287.435394(ms)
GF-CPU	164598.122304(ms)
GF-SHARED-1	245.738472(ms)
GF-SHARED-N	171.329283(ms)
IMAGESET-DEFAULT	1298.983240(ms)
IMAGESET-NONDEFAULT-M	641.025059(ms)

[1] GF

수행 시간
GF-CPU << GF-GLOBAL < GF-SHARED-1 < GF-SHARED-N

예상한 대로 결과가 나왔다. CPU에서 연산을 수행한 GF-CPU는 수행 시간이 비교가 힘들 정도로 매우 크게 나왔고, GPU에서 연산을 수행한 세 함수는 GF-GLOBAL, GF-SHARED-1, GF-SHARED-N 순서대로 수행 시간이 빨라졌다.

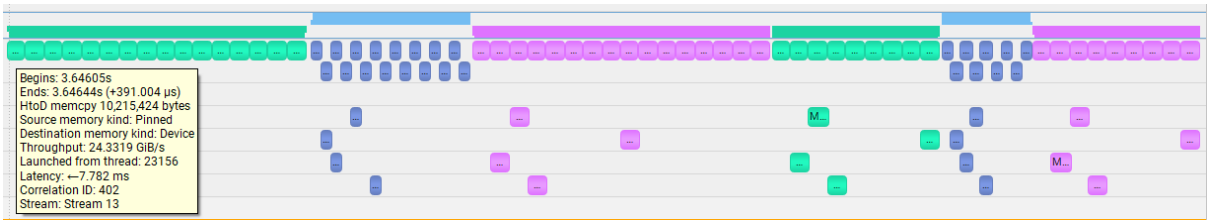
GF-CPU	CPU에서는 SIMT 연산을 병렬화 할 수 없다. 결국 큰 이미지에 존재하는 모든 픽셀(예시의 경우 2,553,856개)에 대해 serial 하게 연산을 수행해야 하므로, 수행 시간이 매우 크다.
GF-GLOBAL	병렬화를 통해 GF-CPU보다는 훨씬 빠른 수행 시간을 보이지만, global memory에 존재하는 이미지에 대해 load/store 연산을 수행하기 때문에 오버헤드가 크다.
GF-SHARED-1	해당 쓰레드 블록에 대해, 필요한 정보를 모두 shared memory에 미리 load 하여 사용한다. 이 덕분에 global memory operation이 크게 줄어 GF-GLOBAL보다 빠른 수행 시간을 얻을 수 있다. 하지만, shared memory에 값을 저장하는 과정에서 branching이 상당히 많이 일어났고, 이로 인해 warp divergence 오버헤드가 발생했을 것으로 추측한다.
GF-SHARED-N	GF-SHARED-1에 MWpT(More Work per Thread) 기법을 적용한 버전으로, 쓰레드 생성 오버헤드를 최소화하기 위해 하나의 쓰레드가 여러 일을 동시에 하도록 하였다. 쓰레드 블록 사이즈가 16*16일 때 WPT(Work per Thread)를 4로 선정한 이유는, 그렇게 해야 shared memory의 타일에 값을 불러올 때 하나의 쓰레드가 네 가지 이상의 일을 하지 않기 때문이다. (외곽의 픽셀을 추가로 144개 저장해야 하는데, 쓰레드 블록 내의 쓰레드가 64개 미만인 경우 네 번의 스텝을 거쳐야 타일을 완성할 수 있다.) 실제로 쓰레드 생성 오버헤드가 컸는지, GF-SHARED-1에 비해 유의미한 시간 단축을 할 수 있었다.

[2] IMAGESET

수행 시간
IMAGESET-DEFAULT ≒ 2 * IMAGESET-NONDEFAULT-M

역시 예상한 대로 결과가 나왔다. 비동기 연산을 이용해 디바이스 연산(H2D->KERNEL->D2H)을 overlapping 한 IMAGESET-NONDEFAULT-M이 일반적인 동기 연산을 사용한 IMAGESET-DEFAULT보다 두 배 정도 빨랐다.

Nsight를 이용해 타임라인을 추적할 수 있다. IMAGESET-DEFAULT의 경우, pageable memory(host) 영역에 존재하는 값을 복사하고 있고, default stream 만을 이용해 연산이 동기적으로 수행되고 있다.



반면 IMAGESET-NONDEFAULT-M의 경우 pinned memory(host) 영역에 존재하는 값을 복사하고 있고, 16개의 non-default stream을 이용해 연산을 비동기적으로 처리하고 있다. 우선 pageable memory보다 pinned memory 영역의 값을 복사하는 것이 수행 시간에 있어 훨씬 빠르고(약 세 배), non-default stream을 여러 개 이용하여 연산을 overlapping 하였기 때문에 한 번에 여러 커널 호출을 수행할 수 있다. 사용하는 스트림의 개수는 실험적으로 얻어냈는데, 영상 길이가 충분히 길 때 8~16개를 운용하는 것이 가장 효율이 좋았다. H2D, D2H 복사나 커널 연산 자체가 그렇게 오래 걸리지 않기 때문에, 스트림의 개수가 많아도 충분히 활용하지 못하는 것 같았다.

이번엔 사진 크기에 따른 GF 커널의 속도 차이이다.

Image size	1856x1376	7360x4832	9984x6400
GF-GLOBAL	21.221701(ms)	287.435394(ms)	492.392834(ms)
GF-CPU	11938.703125(ms)	164598.122304(ms)	294634.832524(ms)
GF-SHARED-1	20.239842(ms)	245.738472(ms)	432.192932(ms)
GF-SHARED-N	15.294934(ms)	171.329283(ms)	310.083231(ms)

Image size	1856x1376	7360x4832	9984x6400
GF-GLOBAL	0.0083(us/pixel)	0.0081(us/pixel)	0.0077(us/pixel)
GF-CPU	4.6747(us/pixel)	4.6283(us/pixel)	4.6110(us/pixel)
GF-SHARED-1	0.0079(us/pixel)	0.0068(us/pixel)	0.0067(us/pixel)
GF-SHARED-N	0.0059(us/pixel)	0.0048(us/pixel)	0.0048(us/pixel)

위의 표는 수행 시간(ms)을, 아래의 표는 픽셀 당 처리 속도(us/pixel)을 나타낸다. 아주 근소한 차이지만, 픽셀 당 처리 속도가 사진 크기와 관계 없이 일정한 GF-CPU에 비해, GPU 디바이스를 사용한 GF-GLOBAL, GF-SHARED-1, GF-SHARED-N의 경우 사진이 커질수록 픽셀 당 처리 속도가 빨라지는 것을 확인할 수 있다. 따라서, 가로 세로 길이가 큰 사진 및 영상 처리를 할 땐 CUDA를 이용해 병렬 처리 하는 것이 절대적인 수행 속도와 연산량 감소 방면에 있어 유리하다.