GPU Computing Project

이름 : 김민교

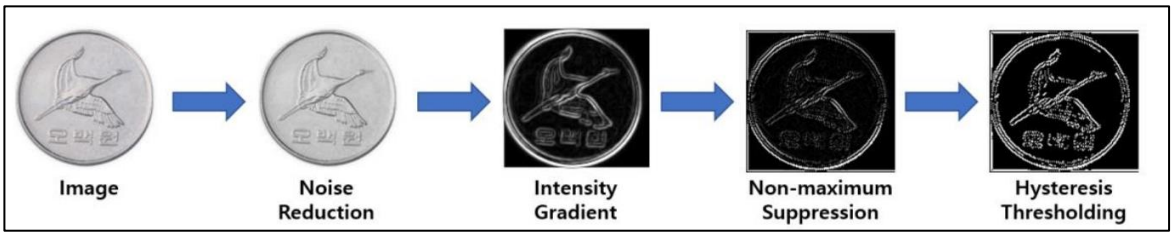
학번 : 2018204058

담당 교수님 : 공영호 교수님

과목명 : GPU Computing

1. Introduction

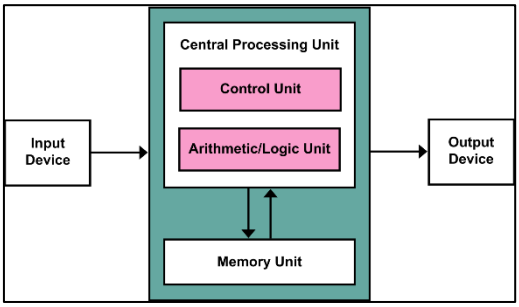
Canny Edge Detection을 GPU를 이용해 수행한다. Canny Edge Detection은 이미지의 Edge를 검출하는 알고리즘이다. 이 알고리즘은 한 이미지를 가지고 연산을 4번한다. Gray Scale, Noise Reduction, Intensity Gradient, Non-maximum Suppression, Hysteresis Thresholding 이 그 연산들이다 (Figure 1). 이미지의 픽셀을 일일 계산해야 하므로, 이미지가 커질수록 CPU에서 하면 속도가 느려진다. 그러므로 GPU로 이미지의 픽셀을 병렬적으로 계산해 Canny Edge Detection을 가속화한다. 또한 직접 GPU programming을 진행하면서 GPU 메모리 관리, 메모리 접근과 블록과 그리드의 차원 등을 탐구하고, 이들을 어떻게 적합하게 사용할지 체득한다.



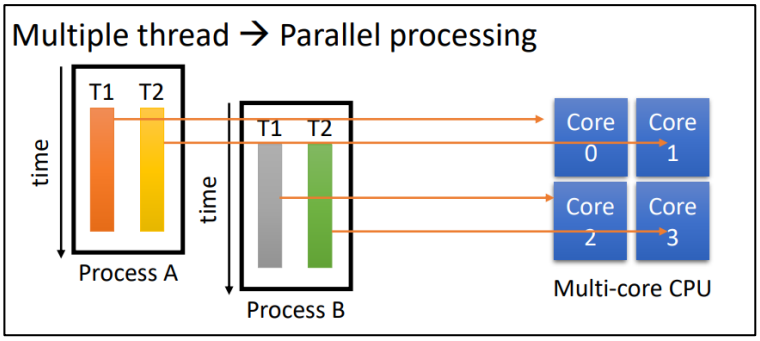
2. Background

A. CPU 구조

CPU는 Central Processing Unit의 약자다. CPU는 폰 노이만 구조에 기반하여 발전해왔다. 이 구조는 명령어가 순차적으로 실행되어야 하는데, 이것이 문제점을 많이 일으킨다. 이른바 "von Neumann Bottleneck"이라는 것이다. 모든 계산의 결과가 ALU 연산을 거쳐 반드시 메모리 어딘가에 저장되어야 한다. 그렇기 때문에 항상 한번에 하나의 명령어를 수행한다.

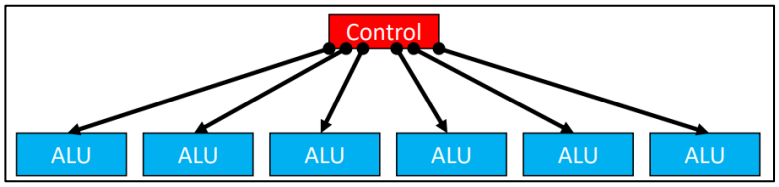


이런 CPU의 단점을 해결하고자 많은 방법이 나왔는데 그 중 하나가 멀티 코어 CPU다(Figure 4). CPU의 코어를 여러 개를 두어 한 번에 스레드를 여러 개 수행한다. 그러나 CPU의 코어는 한 칩에 2개에서 8개정도 밖에 되지 않는다.



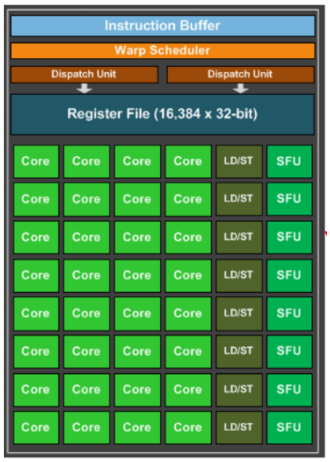
B. GPU 구조

GPU는 Graphics Processing Unit의 약자다. GPU의 원래 목적은 3D 그래픽 처리이다. GPU는 병렬 계산에 특화되어 있다. GPU의 한 칩에 1024개에서 4096개의 코어를 가지 고 있다. 이는 코어 하나가 스레드 하나를 차지해서 동시에 1024개에서 4096개를 동시에 수행할 수 있다는 의미다.

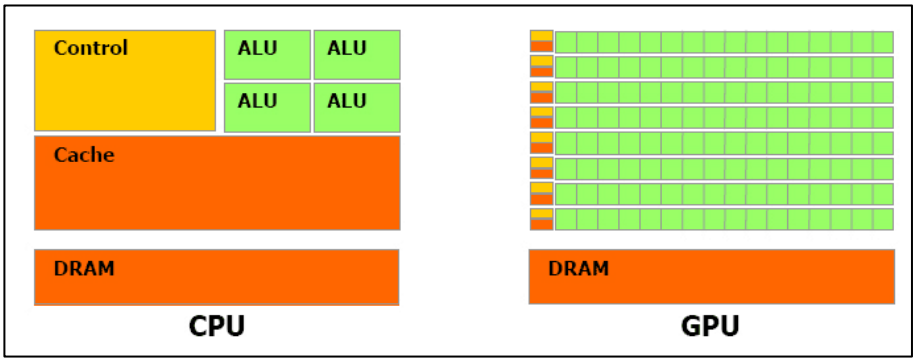


GPU는 CPU와 다르게 여러 개의 ALU와 하나의 Control unit을 가지고 있다. 하나의 명령으로 같은 연산을 동시에 할 수 있다.

GPU에는 SM과 SP가 있다. SP에는 블록 단위로 맵핑된다. 블록의 스레드들이 SM 내부에 있는 SP들로 맵핑되서 작업을 수행한다. SP는 보통 core라고 이야기 한다. SP가 하나의 스레드를 담당하여서 연산을 한다. 보통 각 SM에 32개의 SP와 8개의 SFU, 2개의 TEX로 구성된다. SM이 많을수록 동시에 처리할 수 있는 양이 많아지며, 그에 따라 시간도 많이 단축된다. SM은 Warp 단위로 스케줄링을 진행한다. Warp는 32개의 스레드 집합을 의미한다. Warp에 있는 모든 스레드들은 동일한 연산을 수행한다. SM에는 한 개의 컨트롤 로직밖에 없다. SM의 모든 코어(=SP)가 동일한 컨트롤 로직에 의해 동작한다.



C. CPU 대비 GPU에서의 연산 시 장단점



CPU는 복잡한 연산을 하지만 코어의 수가 적다. GPU는 CPU에 비해 단순한 연산을 하지만 코어의 수가 많다. CPU가 여러 개의 코어로 병렬적으로 수행한다 해도 수만 개의 코어를 가진 GPU보다는 당연히 느릴 수밖에 없다. 동일한 계산을 수많은 개체에 해야 하는 경우에 GPU를 사용하는 게 효율적이다. 그러나 GPU에서 글로벌 메모리에 접근하려면 많은 비용이 든다. 그래서 쉐어드 메모리를 많이 이용해야 하는데, 이를 프로그래밍 하려면 많이 복잡하다. 또한 연

산을 병렬적으로 수행해야 하니 Race Condition이 발생할 수도 있다. 이런 경우도 프로그래머가 많은 것을 고려해서 프로그래밍 해야 할 필요가 있다.

3. Code explanation & Optimization

A. Device Query 확인하기

[전체 결과]

```
Device 0: "Tesla T4"
  CUDA Driver Version / Runtime Version          11.2 / 11.2
  CUDA Capability Major/Minor version number:    7.5
  Total amount of global memory:                 15110 MBytes (15843721216 bytes)
  (40) Multiprocessors, ( 64) CUDA Cores/MP:     2560 CUDA Cores
  GPU Max Clock rate:                            1590 MHz (1.59 GHz)
  Memory Clock rate:                             5001 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 4194304 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        65536 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1024
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 3 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 0 / 4
  Compute Mode:
```

[중요 정보]

```
  (40) Multiprocessors, ( 64) CUDA Cores/MP:     2560 CUDA Cores
```

SM은 40개, 한 SM 당 코어는 64개가 있다. 총 2560개의 코어가 존재한다.

```
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1024
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
```

Warp size는 32개이다. 한 SM 당 가질 수 있는 최대 스레드의 개수는 1024개, 블록이 가질 수 있는 최대 스레드는 1024개다.

CUDA Capability version number 는 7.5이다. 이는 SM당 32개의 block이 허용된다. 총 40개의 SM이 있으니, 40 * 32 = 1280개. 총 1280개의 블록까지는 괜찮게 최적화가 될 것 같다.

강의 교안 5장에 따르면 SM이 32개의 SP를 가지고 있을 때, global memory access에 따른 지연 시간을 줄이려면 최소 25개의 Warp가 필요하다고 한다. 그런데 디바이스 쿼리를 통해 한 SM이 64개의 SP를 가지고 있으니 최소 50개의 Warp가 필요하다. 즉 한 SM 당 32*50 =1600, 최소 1600개의 스레드가 존재해야 한다.

B. 전역 변수와 constant 변수 선언

```
10  //declare constant memory
11  __constant__ int d_WIDTH;       // 이미지 너비
12  __constant__ int d_HEIGHT;      // 이미지 높이
13  __constant__ float filter[25];  // Noise Reduction에 쓰이는 필터
14  __constant__ int filter_x[9];   // Intensity Gradient에 쓰이는 x 필터
15  __constant__ int filter_y[9];   // Intensity Gradient에 쓰이는 y 필터
16  __constant__ uint8_t low_t;     // Hysteresis_Thresholding에 쓰임
17  __constant__ uint8_t high_t;    // Hysteresis_Thresholding에 쓰임
```

line 11~12 : 이미지 너비와 이미지 높이는 Canny Edge Detection을 수행하면서 스레드들이 필요로 하는 값들이다.

line 13 ~ 15 : filter는 Noise Reduction을 수행할 때 모든 스레드들이 읽어야 하는 값이다. filter_x와 filter_y는 Intensity Gradient를 수행할 때 모든 스레드들이 읽어야 하는 값이다.

line 16 ~ 17 : Hysteresis_Thresholding을 수행할 때 모든 스레드들이 읽어야 하는 값이다.

이 변수들은 전부 바뀌지 않고 read만 할 뿐 write는 하지 않기 때문에 __constant__ 메모리에 올렸다. 또한 모든 스레드들이 계속 읽어야 한다. 그래서 __global__ 메모리에 올리기보단 __constant__메모리에 올려서 접근 시간을 줄이고자 하였다.

```
19  uint8_t * d_gray;              // GPU에서 grayscale한 이미지 참조
20  uint8_t * d_gaussian;          // GPU에서 gaussian한 이미지 참조
21  uint8_t * d_sobel;             // GPU에서 soble 적용한 이미지 참조
22  uint8_t * d_angle;             // GPU에서 angle 정보 참조
23  uint8_t * d_suppression_pixel; // GPU에서 suppression_pixcel 적용한 이미지 참조
```

Canny Edge Detection은 각 연산이 독립적으로 이루어지는 게 아니라 종속된다. GrayScale한 결과에 Gaussian을 적용하고, Gaussian을 적용한 결과에 sobel을 적용한다. sobel을 적용한 결과에 suppression_pixcel을 적용한다. 이렇게 이전 연산에 쓰인 값이 재활용된다. 그래서 전역 변수로 두어서 다음 연산을 하는 함수에서도 참조가 가능케 하였다. 이렇게 재활용 되는 값들을 메모리에 놔두면 매 연산마다 cudaMemcpy와 cudaMalloc 호출을 덜 하게 된다. 그래서 cudaMemcpy를 할 때, 글로벌 메모리에 접근하게 된다. 이런 과정을 줄이면서 시간을 줄였다.

C. GPU_Grayscale

```
47  void GPU_Grayscale(uint8_t* buf, uint8_t* gray, uint8_t start_add, int len) {
48
49      //buf를 GPU에 올릴려면 +2해줘서 빠지는게 없이 올라가야한다.
50      int realLen = len+2;
51      // 메모리 할당할 size
52      int size = realLen-start_add;
53
54      // 과제에서 result 부분, gray scale한거 저장할 메모리 할당
55      cudaMalloc((void**)&d_gray,size);
56
57      // 원본 사진을 d_gray에 옮긴다. 원본으로 gray를 만들 수 있다.
58      cudaMemcpy(d_gray,buf+start_add,size,cudaMemcpyHostToDevice);
59
60      // launch kernel function
61      grayScaleKernel <<< 782, 1024>>>(d_gray);
62
63      //GPU에서 작업한 gray 호스트로 전달.
64      cudaMemcpy(gray+start_add,d_gray,size,cudaMemcpyDeviceToHost);
65  }
```

line 51 : Canny.cu에서 len-2값을 전달해준다. 그래서 이 함수에서는 len+2를 했다.

line 53 : size만큼 디바이스에 메모리 공간을 할당하고 d_gray가 가리키게 한다. cudaMemcpy()를 통해 d_gray가 가리키는 공간에 원본 이미지(buf)를 옮긴다. GrayScale 연산은 한 픽셀이 다른 픽셀에 영향을 끼치지 않아서 원본의 정보로 gray를 만들고 바로바로 변경사항을 저장해도 된다.

[grayScaleKernel 함수]

```
26  __global__ void grayScaleKernel(uint8_t* d_gray){
27
28      // 스레드 인덱스
29      int thread_idx = blockIdx.x* 1024+threadIdx.x;
30      // 배열의 인덱스, 스레드가 배열의 요소3개씩 처리한다.
31      int arr_idx = (thread_idx*3);
32      if(arr_idx >2400051)
33          return;
34
35      //BGR. d_buf[i]= B, d_buf[i+1] = G, d_buf[i+2]=R
36      float B = d_gray[arr_idx] * 0.114;
37      float G = d_gray[arr_idx +1] * 0.587;
38      float R = d_gray[arr_idx +2] * 0.299;
39
40      int tmp = B+G+R;
41
42      d_gray[arr_idx] =tmp;
43      d_gray[arr_idx+1] = tmp;
44      d_gray[arr_idx+2] = tmp;
45  }
```

line 32 : 스레드를 픽셀 수보다 많이 만들었기 때문에 배열 범위를 벗어나면 연산을 하지 않는

다.

나머지 코드는 CPU_Func.cu와 동일하다.

Kernel의 연산을 최적화하기 위해서 사용한 방법 :

처음 작성하였을 때는 buf의 정보도 디바이스 메모리 공간에 올렸었다. 그리고 buf의 B,G,R을 가져와서 계산 후에 d_gray에 저장하였다. 이 방식은 global memory에 size만큼 CudaMemcpy()와 CudaMalloc()을 해야한다. CudaMemcpy()함수를 호출하면서 자동적으로 글로벌 메모리에 더 많이 접근하게 된다. buf의 정보를 d_gray에 그대로 옮겨서 글로벌 메모리의 접근을 줄였다.

그리고 이 d_gray는 다음 연산인 Noise_Reduction 함수에서 쓰기 때문에 메모리 해제를 하지 않고 디바이스에 그대로 남겨두었다. 이렇게 되면 Noise_Reduction에서 grayScale한 이미지에 대하여CudaMemcpy와 CudaMalloc을 수행하지 않아도 된다.

코드에서 Block 및 Thread의 수를 정한 이유 :

Block은 782개, Block의 Thread는 최대값인 1024로 정했다.

colab에서 할당받은 GPU의 한 SM은 32 * 40 = 1280개의 Block을 수용할 수 있다. GrayScale에서는 블록 안의 스레드들끼리 공유할 메모리도 없다. 그래서 블록이 가질 수 있는 최대 스레드 개수 1024로 할당했다. 블록은 782개를 선언했다. SM이 40개니 한 SM당 약 20개의 블록을 가져가서 계산할 것이다. 20 * 1024 = 20480개의 스레드가 할당될 것이다. 이렇게 되면 글로벌 메모리에 접근할 때에 대비하여 필요한 최소 스레드 개수도 넘어가니 최적화가 될 것이라 생각했다.

D. GPU_Noise_Reduction

```
120 void GPU_Noise_Reduction(int width, int height, uint8_t *gray, uint8_t *gaussian) {
121
122     //---------- width와 height를 컨스턴트 메모리에 올린다.----------
123     cudaMemcpyToSymbol(d_WIDTH,&width, sizeof(int));
124     cudaMemcpyToSymbol(d_HEIGHT,&height, sizeof(int));
125
126
127    //---------- filter를 계산.  25번이니깐 cpu에서 돌렸다.----------
128     float h_filter[25] = {0};
129     float sigma = 1.0;
130     for (int i = -2; i <= 2; i++) {
131         for (int j = -2; j <= 2; j++) {
132             h_filter[(i + 2) * 5 + j + 2]
133                 = (1 / (2 * 3.14* sigma * sigma)) * exp(-(i * i + j * j) / (2 * sigma * sigma));
134         }
135     }
136
137     //---------- 필터를 컨스턴트 메모리에 올린다.----------
138     cudaMemcpyToSymbol(filter,&h_filter, sizeof(float)*25);
```

line 123 ~124 : 전달 받은 width와 height를 컨스턴트 메모리에 올린다. 이 값은 canny Edge detection을 수행하면서 많이 참조하고 값을 바꾸지 않기 때문에 컨스턴트 메모리에 올렸다.

line 128 ~ 138 : 필터를 만든다. 25번 반복이기 때문에 GPU에 올리는 것보다 CPU에서 해결하는 것이 더 빠르다고 판단했다. GPU에서 계산하려면 메모리에 참조하는 시간이 더 길 것이라 판단했다. 이 필터를 컨스턴트 메모리에 올린다. Noise_Reduction을 수행하면서 모든 스레드들이 읽어와야 하기 때문에 빠르게 접근할 수 있는 컨스턴트 메모리에 올렸다.

```
146     //----------  Grid, Block 차원 결정 ----------
147     const int TILE_WIDTH  = 50;
148     const int TILE_HEIGHT = 20;
149     dim3 dimGrid(width/TILE_WIDTH,height/TILE_HEIGHT,1); //2차원, Block 개수 (20,40), 800개
150     dim3 dimBlock(TILE_WIDTH,TILE_HEIGHT,1);  // 2차원, Block안의 스레드 개수 (50,20) 1000개
151
152     //---------- 커널 함수 실행 ----------
153     Noise_Reduction_Kernel <<<dimGrid, dimBlock,TILE_HEIGHT*TILE_WIDTH*sizeof(uint8_t) >>>(d_gaussian,d_gray);
154
155     ///---------- GPU에서 작업한 가우시안 호스트로 전달.----------
156     cudaMemcpy(gaussian,d_gaussian,width*height*3,cudaMemcpyDeviceToHost);
157
158     //---------- 메모리해제.----------
159     cudaFree(d_gray);
160     cudaFree(filter);
161 }
```

line 146~150 : 그리드, 블록 차원 수 결정

line 153 : 커널 함수 실행, 커널 함수에서 사용할 shared memory의 배열 크기를 밖에서 명시해주었다.

line 156 : GPU에서 작업한 것을 호스트로 전달

line 159~160 : d_gray와 filter를 이제 사용하지 않으니 메모리 해제한다.

[Noise_Reduction_Kernel 함수]

```
69  __global__ void Noise_Reduction_Kernel(uint8_t *gaussian,uint8_t *gray){
70
71      int row = blockIdx.y*blockDim.y+threadIdx.y;
72      int col = blockIdx.x*blockDim.x+threadIdx.x;
73
74      //global index
75      int global_idx = row * d_WIDTH + col;
76
77      extern __shared__ uint8_t pixcel[];
78
79      pixcel[threadIdx.y*blockDim.x + threadIdx.x] = gray[global_idx*3];
80      __syncthreads();
```

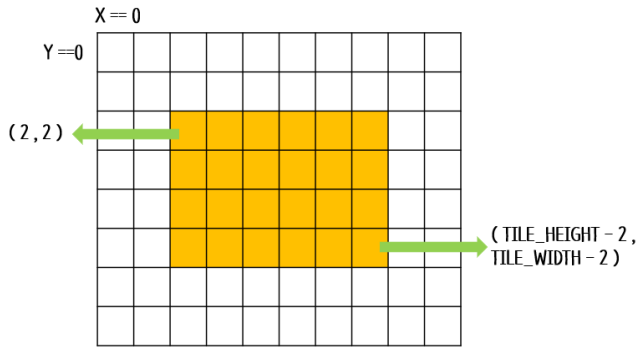
line 75 : 블록 인덱스, 이미지 WIDTH, 스레드 인덱스를 통해 글로벌 인덱스를 구한다.

line 79 : Noise Reduction을 수행할 때에, 픽셀들을 중복되게 접근한다. 이런 픽셀들을 글로벌 메모리에서 가져올 것이 아니라, 쉐어드 메모리에서 가져오면 좀 더 시간을 단축할 수 있다. 그래서 픽셀 배열에 현재 블록의 픽셀 정보를 담는다.

line 80 : Race Condition을 방지하기 위해 __synchthreads()를 호출한다.

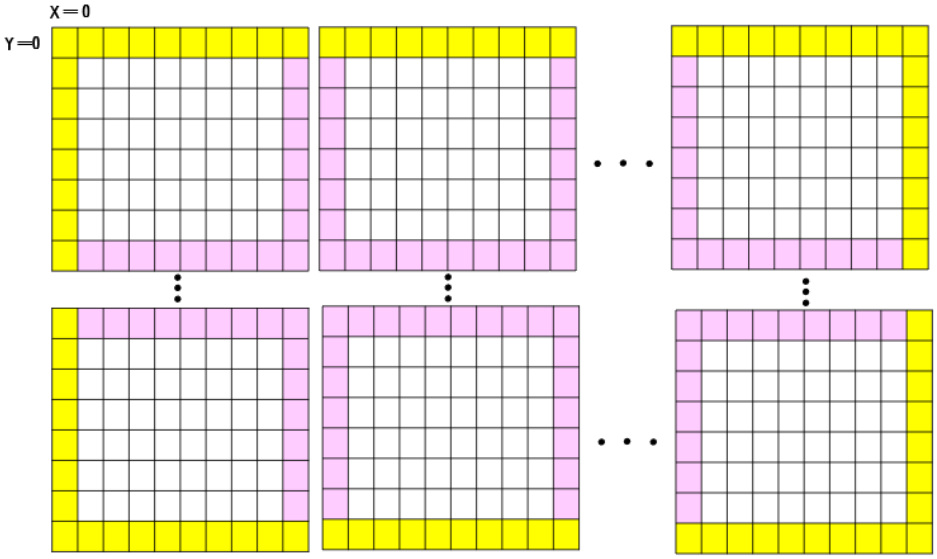
```
84      // shared memory로만 계산이 가능한 경우
85      if( (threadIdx.x >= 2 && threadIdx.x <=blockDim.x - 3) && (threadIdx.y>=2 && threadIdx.y <= blockDim.y -3) )
86      {
87          //GaussianBlur
88          //25번 반복정도는 스레드가 혼자 하기.
89          for(int i =-2; i<=2; i++ ){
90              for(int j= -2;j<=2 ;j++){
91                  v+= pixcel[ ((i+threadIdx.y)*(blockDim.x)+(j+threadIdx.x))] * filter[(i+2)*5+(j+2)];
92              }
93          }
94          gaussian[global_idx* 3] = v;
95          gaussian[global_idx* 3 +1] = v;
96          gaussian[global_idx* 3 +2] = v;
97      }
```

line 85 ~ 97 : 이 조건문에 걸리는 픽셀을 계산할 때에는 shared memory만 접근해도 되는 경우이다.




```
99      // global memory를 써야하는 경우와 제로 패딩한 값이 필요할 때,
100     else{
101         for(int i =-2; i<=2; i++ ){
102             for(int j= -2;j<=2 ;j++){
103                 if( i+row < 0 || i+row >=d_HEIGHT || j+col <0 || j+col>=d_WIDTH)
104                     v+=0.0;
105                 else if(i+threadIdx.y < 0 || i+threadIdx.y>=blockDim.y || j+threadIdx.x <0 || j+threadIdx.x>=blockDim.x){
106                     v+= gray[ ((i+row)*(d_WIDTH)+(j+col)) *3] * filter[(i+2)*5+(j+2)];
107                 }
108                 else
109                     v+= pixcel[ ((i+threadIdx.y)*(blockDim.x)+(j+threadIdx.x))] * filter[(i+2)*5+(j+2)];
110             }
111         }
112         gaussian[global_idx* 3] = v;
113         gaussian[global_idx* 3 +1] = v;
114         gaussian[global_idx* 3 +2] = v;
115     }
116 }
```

line 100 ~ 116: 이 조건문에 걸리는 픽셀을 계산할 때에는 값이 shared memory에 존재 하지 않아 글로벌 메모리에서 값을 얻어오거나 0을 더할 때이다.



그림과 같이 핑크색이 칠해져 있는 곳은 shared memory로만 계산할 수 없다. 다른 블록의 픽셀 값이 필요하다. 이럴 때에는 글로벌 메모리에서 접근한다. 또 노란색이 칠해져 있는 곳은 원래는 제로 패딩이 되어 0을 더해야 하는 경우가 있는 곳이다. 실제론 필터가 5x5라서 더 칠해져야 하지만 간략하게 설명하기 위해 한 줄만 칠했다.

Kernel의 연산을 최적화하기 위해서 사용한 방법 :

같은 블록안의 스레드에서 계산할 때 사용되는 값이 중복되는 경우가 많다. 그래서 이런 중복 값들을 글로벌 메모리에서 가져올 것이 아니라, 쉐어드 메모리에서 가져오게 하여 시간을 줄였다.

또한 CPU_Func.cu에서는 제로 패딩을 한 값을 이용해서 필터 연산을 수행한다. 그러나 제로 패딩한 이미지를 얻으려면 스레드를 또 호출해서 값을 넣어야 한다. 이 때에 또 글로벌 메모리에 많이 접근해야 한다. 이런 시간을 줄이기 위해서 제로 패딩한 이미지를 따로 얻지 않고 원본 이미지에서 필터를 계산했다.

코드에서 Block 및 Thread의 수를 정한 이유 :

TILE_WIDTH = 50, TILE_HEIGHT =20으로 정했다. width 10000, height는 800이다.

블록의 개수 : (10000/50 , 800/20) == (200,40) == 800개

한 블록의 스레드 개수 : (50,20) == 1000개

블록이 가질 수 있는 최대 스레드 개수와 비슷하게 1000개로 정했다. 쉐어드 메모리에 1000 bytes가 차지하게 되는데 이는 한 블록당 쉐어드 메모리가 49152 bytes라서 부담이 없다고 생각했다. 블록 안의 스레드들이 최대한 많은 픽셀 값들을 공유하는 것이 속도를 향상케 할 것이라 생각했다.

E. GPU_Intensity_Gradient

```
242  void GPU_Intensity_Gradient(int width, int height, uint8_t* gaussian, uint8_t* sobel, uint8_t*angle){
243      //---------- filter를 정의 ----------
244      int c_filter_x[9] = {-1,0,1
245                          ,-2,0,2
246                          ,-1,0,1};
247      int c_filter_y[9] = {1,2,1
248                          ,0,0,0
249                          ,-1,-2,-1};
250
251      //---------- 필터를 컨스턴트 메모리에 올린다.----------
252      cudaMemcpyToSymbol(filter_x,&c_filter_x, sizeof(int)*9);
253      cudaMemcpyToSymbol(filter_y,&c_filter_y, sizeof(int)*9);
254
```

line 243 ~253 : 필터를 정의한다. 필터는 Intensity_Gradient를 수행하면서 많이 참조하고 값을 바꾸지 않기 때문에 빠르게 접근할 수 있는 컨스턴트 메모리에 올렸다.

```
256      //----------  디바이스 메모리에 할당.----------
257      cudaMalloc((void**)&d_sobel,width*height*3); // *과제에서 source 부분, 원본 이미지 할당.
258      cudaMalloc((void**)&d_angle,width*height);
259
260      //---------- 디바이스에 gaussian 이미지 복사 ----------
261      cudaMemcpy(d_gaussian,gaussian,width*height*3,cudaMemcpyHostToDevice); // source 부분을 GPU에 옮긴다.
262
263
264      //----------  Grid, Block 차원 결정 ----------
265      const int TILE_WIDTH  = 50;
266      const int TILE_HEIGHT = 20;
267      dim3 dimGrid(width/TILE_WIDTH,height/TILE_HEIGHT,1); //2차원, Block 개수 (20,40), 800개
268      dim3 dimBlock(TILE_WIDTH,TILE_HEIGHT,1);  // 2차원, Block안의 스레드 개수 (50,20) 1000개
269
270      //---------- 커널 함수 실행 ----------
271      Intensity_Gradient_Kernel<<< dimGrid,dimBlock,TILE_HEIGHT*TILE_WIDTH*sizeof(uint8_t) >>>(d_gaussian,d_sobel,d_angle);
272
273      ///---------- GPU에서 작업한 sobel과 angle 호스트로 전달.----------
274      cudaMemcpy(sobel,d_sobel,width*height*3,cudaMemcpyDeviceToHost);
275      cudaMemcpy(angle,d_angle,width*height,cudaMemcpyDeviceToHost);
276
277      //---------- 메모리해제.----------
278      cudaFree(d_gaussian);
279      cudaFree(filter_x);
280      cudaFree(filter_y);
```

line 257 ~ 258 : 디바이스에 sobel과 angle값이 올라갈 메모리를 할당하고 d_sobel과 d_angle이 가리키게 한다.

line 261 : gaussian 이미지는 cudaMemcpy()를 호출해야 한다. GPU에서 연산한 gaussian과 CPU에서 gaussian이 floating point 연산 방식이 달라서 좀 차이가 있는 듯 하다. 정확한 계산을 위해 호스트가 제공해주는 gaussian을 사용한다.

line 265~268 : 그리드, 블록 차원 수 결정

line 271 : 커널 함수 실행, 커널 함수에서 사용할 shared memory의 배열 크기를 밖에서 명시해주었다.

line 274~275 : GPU에서 작업한 것을 호스트로 전달

line 278~280 : d_gaussian과 filter_x, filter_y를 이제 사용하지 않으니 메모리 해제한다.

[Intensity_Gradient_Kernel 함수]

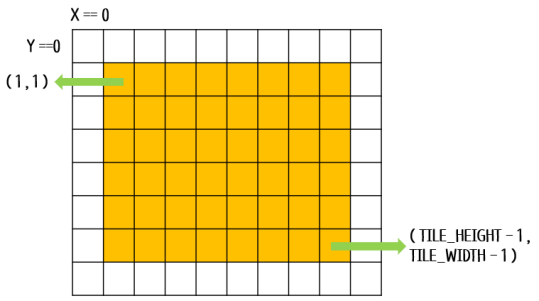
```
164  __global__ void Intensity_Gradient_Kernel(uint8_t* gaussian, uint8_t* sobel, uint8_t* angle){
165
166      int row = blockIdx.y*blockDim.y+threadIdx.y;
167      int col = blockIdx.x*blockDim.x+threadIdx.x;
168
169      //global index
170      int global_idx = row * d_WIDTH + col;
171
172
173      ///---------- 블럭 안의 스레드들이 공유할 픽셀 정보----------
174      extern __shared__ uint8_t pixcel[];
175      pixcel[threadIdx.y* blockDim.x + threadIdx.x] = gaussian[global_idx*3];
176       __syncthreads(); // 모든 스레드들이 픽셀 배열에 값을 넣을 때까지 대기
```

line 170 : 블록 인덱스, 이미지 WIDTH, 스레드 인덱스를 통해 글로벌 인덱스를 구한다.

line 174~175 : Intensity를 수행할 때에, 픽셀들을 중복되게 접근한다. 이런 픽셀들을 글로벌 메모리에서 가져올 것이 아니라, 쉐어드 메모리에서 가져오면 좀 더 시간을 단축할 수 있다. 그래서 픽셀 배열에 현재 블록의 픽셀 정보를 담는다.

line 176 : Race Condition을 방지하기 위해 __synchthreads()를 호출한다.

```
178      int gx = 0;
179      int gy = 0;
180
181      // shared memory로만 계산이 가능한 경우
182      if( (threadIdx.x >= 1 && threadIdx.x <=blockDim.x -2) && (threadIdx.y>=1 && threadIdx.y <=blockDim.y-2) ){
183          //9번 반복정도는 스레드가 혼자 하기.
184          for(int i =-1; i<=1; i++ ){
185              for(int j= -1;j<=1 ;j++){
186                  gy += pixcel[ ((i+threadIdx.y)*(blockDim.x)+(j+threadIdx.x))] * filter_y[(i+1)*3+(j+1)];
187                  gx += pixcel[ ((i+threadIdx.y)*(blockDim.x)+(j+threadIdx.x))] * filter_x[(i+1)*3+(j+1)];
188              }
189          }
190      }
```

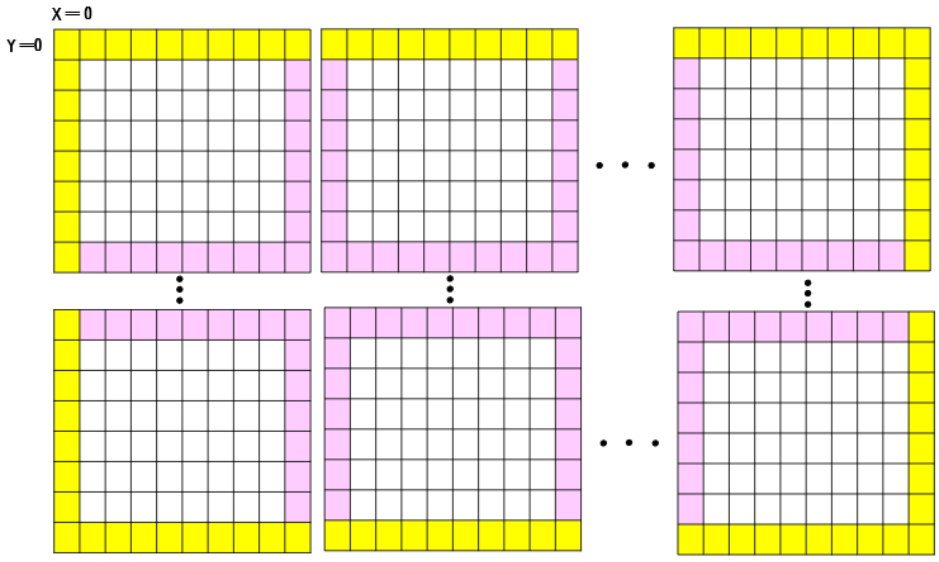


line 182 ~ 190 : 이 조건문에 걸리는 픽셀을 계산할 때에는 shared memory만 접근해도 되는 경우이다. shared memory에 있는 픽셀 값에 접근하여서 연산을 진행한다.

```
191     else{
192         //9번 반복정도는 스레드가 혼자 하기.
193         for(int i =-1; i<=1; i++ ){
194             for(int j= -1;j<=1 ;j++){
195                 if( i+row < 0 || i+row >=d_HEIGHT || j+col <0 || j+col>=d_WIDTH){
196                     //제로 패딩인 경우 0을 더함
197                     gy+=0;
198                     gx+=0;
199                 }
200                 else if(i+threadIdx.y < 0 || i+threadIdx.y>=blockDim.y || j+threadIdx.x <0 || j+threadIdx.x>=blockDim.x){
201                     //쉐어드 메모리로 접근 불가한 경우, 글로벌 메모리에서 해결
202                     float g =gaussian[ ( (row + i) * (d_WIDTH) + col + j)*3];
203                     gy += (int)g * filter_y[(i+1) * 3 + (j+1)];
204                     gx += (int)g * filter_x[(i+1) * 3 + (j+1)];
205                 }
206                 else{
207                     //쉐어드 메모리에서 해결 가능한 경우
208                     gy += pixcel[ ((i+threadIdx.y)*(blockDim.x)+(j+threadIdx.x))] * filter_y[(i+1)*3+(j+1)];
209                     gx += pixcel[ ((i+threadIdx.y)*(blockDim.x)+(j+threadIdx.x))] * filter_x[(i+1)*3+(j+1)];
210                 }
211             }
212         }
213     }
```

line 191~ 213: 이 조건문에 걸리는 픽셀을 계산할 때에는 값이 shared memory에 존재하지 않아 글로벌 메모리에서 값을 얻어오거나 0을 더할 때이다.

line 202 : 조금이라도 글로벌 메모리 접근을 줄이고자, 한번 접근해서 레지스터에 저장한다. 그리고 레지스터에서 값을 읽어와 계산을 한다.



그림과 같이 핑크색이 칠해져 있는 곳은 shared memory로만 계산할 수 없다. 다른 블록의 픽셀 값이 필요하다. 이럴 때에는 글로벌 메모리에서 접근한다. 또 노란색이 칠해져 있는 곳은 원래는 제로 패딩이 되어 0을 더해야 하는 경우가 있는 곳이다.

```
    int t = sqrt(gx * gx + gy * gy);

    uint8_t  v = 0;
    if (t > 255) {
        v = 255;
    }
    else
        v = t;

    sobel[global_idx * 3] = v;
    sobel[global_idx * 3 + 1] = v;
    sobel[global_idx * 3 + 2] = v;

    float t_angle = 0;
    if(gy != 0 || gx != 0)
        t_angle= (float)atan2(gy, gx) * 180.0 / 3.14;
    if ((t_angle > -22.5 && t_angle <= 22.5) || (t_angle > 157.5 || t_angle <= -157.5))
        angle[global_idx] = 0;
    else if ((t_angle > 22.5 && t_angle <= 67.5) || (t_angle > -157.5 && t_angle <= -112.5))
        angle[global_idx] = 45;
    else if ((t_angle > 67.5 && t_angle <= 112.5) || (t_angle > -112.5 && t_angle <= -67.5))
        angle[global_idx] = 90;
    else if ((t_angle > 112.5 && t_angle <= 157.5) || (t_angle > -67.5 && t_angle <= -22.5))
        angle[global_idx] = 135;
}
```

soble과 angle 정보를 저장하는 코드이다. CPU_Func.cu와 동일하다.

Kernel의 연산을 최적화하기 위해서 사용한 방법 :

같은 블록안의 스레드에서 계산할 때 사용되는 값이 중복되는 경우가 많다. 그래서 이런 중복 값들을 글로벌 메모리에서 가져올 것이 아니라, 쉐어드 메모리에서 가져오게 하여 시간을 줄였다.

또한 CPU_Func.cu에서는 제로 패딩을 한 값을 이용해서 필터 연산을 수행한다. 그러나 제로 패딩한 이미지를 얻으려면 스레드를 또 호출해서 값을 넣어야 한다. 이 때에 또 글로벌 메모리에 많이 접근해야 한다. 이런 시간을 줄이기 위해서 제로 패딩한 이미지를 따로 얻지 않고 원본 이미지에서 필터를 계산했다.

코드에서 Block 및 Thread의 수를 정한 이유 :

TILE_WIDTH = 50, TILE_HEIGHT =20으로 정했다. width 10000, height는 800이다.

블록의 개수 : (10000/50 , 800/20) == (200,40) == 800개

한 블록의 스레드 개수 : (50,20) == 1000개

블록이 가질 수 있는 최대 스레드 개수와 비슷하게 1000개로 정했다. 쉐어드 메모리에 1000 bytes가 차지하게 되는데 이는 한 블록당 쉐어드 메모리가 49152 bytes라서 부담이 없다고 생각했다. 블록 안의 스레드들이 최대한 많은 픽셀 값들을 공유하는 것이 속도를 향상케 할 것이라 생각했다.

F. GPU_Non_maximum_Suppression

```
353  void GPU_Non_maximum_Suppression(int width, int height, uint8_t *angle,uint8_t *sobel, uint8_t *sup
354
355      // 디바이스에서 min, max를 가리키는 포인터
356      int * d_min;
357      int * d_max;
358
359      //----------  디바이스 메모리에 할당.----------
360      cudaMalloc((void**)&d_suppression_pixel,width*height*3);
361      cudaMalloc((void**)&d_min, sizeof(int));
362      cudaMalloc((void**)&d_max, sizeof(int));
363
364      //전달 받은 min max값 임시로 저장.
365      int g_min = min;
366      int g_max = max;
367
```

line 356~357 : 디바이스에서 min, max가 할당된 메모리를 가리킬 포인터다.

line 360~362 : 디바이스에 min, max 메모리를 할당하고, d_suppression_pixcel을 저장할 메모리를 할당한다.

line 365~366 : 전달 받은 매개변수를 통해 전달 받은 min, max를 임시로 저장한다. (참조자 사용에 익숙치 않았습니다)

```
368      //----------  Grid, Block 차원 결정 ----------
369      // Block안의 스레드 개수를 적게 하는 것이 시간이 더 짧게 걸렸다.
370      // 아무래도 min, max를 비교할 때 요구되는 반복문의 횟수가 짧아져서 그런듯 깊다.
371      const int TILE_WIDTH  = 25;
372      const int TILE_HEIGHT = 2;
373      dim3 dimGrid(width/TILE_WIDTH,height/TILE_HEIGHT,1); //2차원, Block 개수 1000개
374      dim3 dimBlock(TILE_WIDTH,TILE_HEIGHT,1);  // 2차원, Block안의 스레드 개수 800개
375
376      //---------- 커널 함수 실행 ----------
377      Non_maximum_Suppression_Kernel<<< dimGrid,dimBlock >>>(d_angle,d_sobel,d_suppression_pixel,d_min,d_max);
378
379      //---------- GPU에서 작업한 것 호스트로 전달.----------
380      cudaMemcpy(suppression_pixel,d_suppression_pixel,width*height*3,cudaMemcpyDeviceToHost);
381      cudaMemcpy(&g_min,d_min,sizeof(int),cudaMemcpyDeviceToHost);
382      cudaMemcpy(&g_max,d_max,sizeof(int),cudaMemcpyDeviceToHost);
383
384      //min, max값 업데이트
385      min =g_min;
386      max =g_max;
387
388      //---------- 메모리해제.----------
389      cudaFree(d_sobel);
390      cudaFree(d_angle);
391  }
```

line 371 ~374 : TILE_WIDTH와 TILE_HEIGHT을 결정하고 grid와 block의 차원을 정의한다.

line 377 : suppression을 수행할 커널 함수를 실행한다.

line 380 ~ 382 : GPU에서 작업한 것을 호스트로 전달한다.

line 389~ 390 : d_sobel과 d_angle은 이제 쓰이지 않기 때문에 메모리에서 해제한다.

[Non_maximum_Suppression_Kernel 함수]

```
285  __global__ void Non_maximum_Suppression_Kernel(uint8_t* angle,uint8_t* sobel, uint8_t* s
286
287      int row = blockIdx.y*blockDim.y+threadIdx.y;
288      int col = blockIdx.x*blockDim.x+threadIdx.x;
289
290      // 맨 위, 맨 아래, 맨 왼쪽, 맨 오른쪽인 경우에는 연산하지 않는다.
291      if(row==0 || row == d_HEIGHT-1||col==0 || col==d_WIDTH-1){
292          return;
293      }
294
295      //---------- 블럭 안의 스레드들이 공유할 v값 min, max를 결정하기 위함----------
296      __shared__ uint8_t value_v[50];
297
298      //global index
299      int global_idx = row * d_WIDTH + col;
```

line 291 ~ 293 : 픽셀이 맨 윗줄, 맨 아랫줄, 맨 왼쪽, 맨 오른쪽인 경우에는 연산을 안 한다.

line 296 : 블록안의 각 스레드들이 얻어낸 v 값을 저장하는 배열이다. min, max값을 결정하기 위해 shared memory에 넣고 한꺼번에 비교해야 한다.

```
304      // 조금이라도 글로벌 메모리 접근을 줄이고자
305      // 한번 접근해서 레지스터에 저장. 이 값으로 if문 조건 판별
306      uint8_t tmp_angle = angle[global_idx];
307
308      if (tmp_angle == 0) {
309              p1 = sobel[((row+1) * d_WIDTH + col)*3];
310              p2 = sobel[((row-1) * d_WIDTH + col) * 3];
311      }
312      else if (tmp_angle == 45) {
313          p1 = sobel[((row + 1) * d_WIDTH + col-1) * 3];
314          p2 = sobel[((row - 1) * d_WIDTH + col+1) * 3];
315      }
316      else if (tmp_angle == 90) {
317          p1 = sobel[((row) * d_WIDTH + col+1) * 3];
318          p2 = sobel[((row) * d_WIDTH + col-1) * 3];
319      }
320      else {
321          p1 = sobel[((row + 1) * d_WIDTH + col+1) * 3];
322          p2 = sobel[((row - 1) * d_WIDTH + col-1) * 3];
323      }
324
325      uint8_t v = sobel[(row * d_WIDTH + col) * 3];
326
327      if ((v >= p1) && (v >= p2)) {
328          suppression_pixel[(row * d_WIDTH + col) * 3] = v;
329          suppression_pixel[(row * d_WIDTH + col) * 3 + 1] = v;
330          suppression_pixel[(row * d_WIDTH + col) * 3 + 2] = v;
331      }
```

line 306 : angle[global_idx]를 조건문에서 계속 참조해야 한다. 계속 글로벌 메모리에 접근하지 않고 레지스터에 한번 넣어서 레지스터에서 접근하도록 한다.

나머지 코드는 CPU_Func.cu 의 코드와 똑같다.

```
333     value_v[threadIdx.y*blockDim.y+threadIdx.x] = v;
334     __syncthreads(); // 모든 스레드들이 v값을 채울 때 까지 대기
335
336
337     //첫번째 스레드에서 한 블럭의 min, max 결정
338     if(threadIdx.x == 0){
339         int t_min = 255;
340         int t_max = 0;
341         for (int i = 0; i<50; i++){
342             if(t_min > value_v[i])
343                 t_min=v;
344             if(t_max < value_v[i])
345                 t_max =v;
346         }
347         //모든 블럭에 대해서 min, max 결정
348         atomicMin(min,t_min);
349         atomicMax(max,t_max);
350     }
351 }
```

line 333~334 : value_v 배열에 각 스레드에서 얻은 v값을 저장한다. 모든 스레드가 저장하길 기다린다.

line 338~350 : 첫번째 스레드에서 shared memory에 있는 value_v 배열의 최소값과 최대값을 구한다. 그리고 atomicMin, atomicMax 함수를 통해 모든 블록에서 v의 최대 최소값을 구한다.

Kernel의 연산을 최적화하기 위해서 사용한 방법 :

Kernel에서 sobel의 값으로 min과 max값을 정해야 한다. shared memory를 이용해서 한 블록에서 min과 max를 구한다. 그리고 atomic 연산을 통해서 전체 블록에서의 min과 max를 구했다.

https://mangkyu.tistory.com/85 -> 참고한 사이트

코드에서 Block 및 Thread의 수를 정한 이유 :

TILE_WIDTH = 10, TILE_HEIGHT =10으로 정했다.

블록 안의 스레드들이 적어야 min, max 값을 구할 때 시간이 적게 걸렸다. 아무래도 한 스레드에서 블록 안의 스레드 개수만큼 반복문을 돌려 min과 max를 구해야 하기 때문인 것 같다.

4. GPU_Hystersis_Thresholding

```
457 void GPU_Hysteresis_Thresholding(int width, int height, uint8_t *suppression_pixel,uint8_t *hyste
458     //----------- 전달 받은 min과 max로 low_t와 hiht_t 구하기 -----------
459     uint8_t diff = max - min;
460     uint8_t c_low_t = min + diff * 0.01;
461     uint8_t c_high_t = min + diff * 0.2;
462 
463     //---------- 필터를 컨스턴트 메모리에 올린다.----------
464     cudaMemcpyToSymbol(low_t,&c_low_t, sizeof(uint8_t));
465     cudaMemcpyToSymbol(high_t,&c_high_t, sizeof(uint8_t));
```

line 459 ~ 461 : low_t와 high_t를 구하고 컨스턴트 메모리에 올린다.

```
468     //----------- 디바이스 메모리를 가리킬 변수 선언 -----------
469     uint8_t * d_hysteresis; //hysteresis, 최종 결과 이미지
470     uint8_t * d_tmp_hysteresis; //중간 저장하는 이미지
471 
472    //----------  디바이스 메모리에 할당.----------
473     cudaMalloc((void**)&d_hysteresis,width*height*3); // RGB값이라 *3 해줘야함.
474     cudaMalloc((void**)&d_tmp_hysteresis,width*height); // RGB값 필요 없어서 weight랑 height만
```

line 469~470 : hysteresis 최종 이미지를 가리킬 포인터와 중간 저장 해놓을 tmp_hystersis를 선언한다

line 473~473 : d_hystersis는 RGB값도 다 저장해야해서 픽셀 수 *3을 한다. 반면 tmp_hystersis는 중간 결과만 보면 되는 것이기 때문에 RGB값을 전부 저장할 필요가 없다. 단순히 픽셀 수만큼만 메모리 공간을 확보한다.

```
482     //---------- 커널 함수 실행 ----------
483     Save_Temp_Hystersis_Kernel<<< dimGrid,dimBlock>>>(d_suppression_pixel,d_tmp_hysteresis);
484     //---------- 커널 함수 실행 ----------
485     Hysteresis_Thresholding_Kernel2<<< dimGrid,dimBlock >>>(d_hysteresis,d_tmp_hysteresis);
486 
487     //---------- GPU에서 작업한 것 호스트로 전달.----------
488     cudaMemcpy(hysteresis,d_hysteresis,width*height*3, cudaMemcpyDeviceToHost);
489 
490     //---------- 메모리해제.----------
491     cudaFree(d_suppression_pixel);
492     cudaFree(d_hysteresis);
493     cudaFree(d_tmp_hysteresis);
494 }
```

line 483 : tmp_hystersis 값을 구하는 커널 함수를 수행한다.

line 485 : 최종 hysteresis 값을 구하는 커널 함수를 수행한다.

line 488~ 493 : GPU에서 작업한 것을 디바이스로 옮기고 메모리를 해제한다.

[Save_Temp_Hystersis_Kernel 함수]

```
399  __global__ void Save_Temp_Hystersis_Kernel(uint8_t* suppression_pixel,uint8_t* tmp_hysteresis){
400
401      int row = blockIdx.y*blockDim.y+threadIdx.y;
402      int col = blockIdx.x*blockDim.x+threadIdx.x;
403
404      //global index
405      int global_idx = row *d_WIDTH + col;
406
407      uint8_t v = suppression_pixel[global_idx*3];
408      if (v < low_t) { // 버려
409              tmp_hysteresis[global_idx]=0;
410          }
411          else if (v < high_t) { //어중간~
412              tmp_hysteresis[global_idx]=123;
413          }
414          else { //무조건 edge로 판별.
415              tmp_hysteresis[global_idx]=255;
416          }
417  }
```

이 함수는 CPU_Func.c의 line 204 memcpy(tmp_hystersis…)를 수행한다. 기존 CPU_Fun.c 에서는 메모리 할당은 픽셀 수 *3만큼 한다. 하지만 실제로 RGB 값이 다 필요하진 않다. 그냥 픽셀 수 만큼만 메모리 할당 및 글로벌 메모리에 접근하면 훨씬 시간을 줄일 수 있다.

```
203  ////////////////////Modified in Version3//////////////////////////////
204      memcpy(tmp_hysteresis,hysteresis,sizeof(uint8_t)*width*height*3);
205          for (int i = 0; i < height; i++) {
206          for (int j = 0; j < width; j++) {
207              if(tmp_hysteresis[(i*width+j)*3] == 123){
208                                  Hysteresis_check(width,height,j,i,hysteresis,tmp_hysteresis);
209              }
210          }
211      }
```

CPU_Func.cu의 memcpy(tmp_hysteresis…)부분

[Hysteresis_Thresholding_Kernel 함수]

```
419  __global__ void Hysteresis_Thresholding_Kernel(uint8_t* hysteresis, uint8_t* tmp_hysteresis){
420      int row = blockIdx.y*blockDim.y+threadIdx.y;
421      int col = blockIdx.x*blockDim.x+threadIdx.x;
422
423      //global index
424      int global_idx = row * d_WIDTH + col;
425
426      //반복문을 빠져나오기 위함
427      bool loop_out =false;
428
429      // 255인곳은 255로 저장
430      if (tmp_hysteresis[global_idx]==255){
431          hysteresis[global_idx*3] = 255;
432          hysteresis[global_idx * 3+1] = 255;
433          hysteresis[global_idx * 3+2] = 255;
434      }
```

이 함수는 tmp_hystersis값을 이용해서 최종 결과가 될 hysteresis에 값을 채운다.

line 427 : tmp_hysteresis가 123인 곳에서 반복문을 돌리는데 빨리 빠져나오기 위한 함수다.

line 430 : tmp_hystersiss값이 255일 때, 최종 결과가 될 hysteresis에 값을 넣어준다

```
435     // 123인 곳은 판별
436     else if(tmp_hysteresis[global_idx] == 123){
437         for (int i = row-1; i < row+2; i++) {
438             for (int j = col-1; j < col+2; j++) {
439                 if ((i < d_HEIGHT && j < d_WIDTH) && (i >= 0 && j >= 0)) {
440                     if (tmp_hysteresis[(i * d_WIDTH + j)] == 255) {
441                         hysteresis[global_idx*3] = 255;
442                         hysteresis[global_idx * 3+1] = 255;
443                         hysteresis[global_idx * 3+2] = 255;
444                         loop_out=true; // 255인 곳 한번 발견하면 바로 반복문을 나온다.
445                         break;
446                     }
447                 }
448             if(loop_out)
449                 break;
450             }
451         }
452     }
```

line 436~452 : 픽셀의 값이 123일 때, 주변 값이 255인지 확인한다, 그리고 255의 값이 있으면 최종 결과 hysteresis에 255로 값을 채운다. 이 때 주변 값 중 하나라도 255를 발견하면 무조건 hysteresis값은 255가 된다. 그러니 바로 반복문을 빠져나와서 불필요한 글로벌 메모리 접근과 반복을 줄인다.

```
453     // 255가 아닌 곳은 0으로
454     if (hysteresis[global_idx* 3] != 255) {
455                 hysteresis[global_idx * 3] = 0;
456                 hysteresis[global_idx* 3+1] = 0;
457                 hysteresis[global_idx * 3+2] = 0;
458     }
459 }
```

line 454 ~ 459 : hysteresis가 255가 아닌 곳은 0으로 채운다.

Kernel의 연산을 최적화하기 위해서 사용한 방법 :

CPU_Func.cu에서는 tmp_hystersis를 구할 때 width*height*3만큼 값을 넣어야 한다. tmp_hystersis에서는 RGB값을 다 가져올 필요가 없다. 그래서 width*height만큼만 연산을 해서 글로벌 메모리의 접근하는 횟수를 1/3로 줄였다.

코드에서 Block 및 Thread의 수를 정한 이유 :

TILE_WIDTH = 50, TILE_HEIGHT =20으로 정했다. width 10000, height는 800이다.

블록의 개수 : (10000/50 , 800/20) == (200,40) == 800개

한 블록의 스레드 개수 : (50,20) == 1000개

블록이 가질 수 있는 최대 스레드 개수와 비슷하게 1000개로 정했다.

5. Result

A. 최종 결과

```
Gray_Scale Time                 =       CPU(0.006759)  GPU(0.001380)    Gray_Scale + 10(0.001380)
Noise_Reduction Time            =       CPU(0.085784)  GPU(0.001131)    Noise_Reduction + 10(0.001131)
Intensity_Gradient Time         =       CPU(0.092246)  GPU(0.003752)    Intensity_Gradient + 10(0.003752)
Non-maximum_Suppression Time    =       CPU(0.016321)  GPU(0.001164)    Non-maximum_Suppression + 10(0.001164)
Hysteresis_Thresholding Time    =       CPU(0.024888)  GPU(0.001220)    Hysteresis Thresholding + 10(0.001220)
total_score is 50
execution time is 0.008647
```

0.08647이 나왔다.

B. Non_maximum_Supperssion 커널을 수행할 때 블록의 스레드 수를 늘릴 경우

```
Gray_Scale Time                 =       CPU(0.006512)  GPU(0.001548)    Gray_Scale + 10(0.001548)
Noise_Reduction Time            =       CPU(0.098385)  GPU(0.001203)    Noise_Reduction + 10(0.001203)
Intensity_Gradient Time         =       CPU(0.091578)  GPU(0.003828)    Intensity_Gradient + 10(0.003828)
Non-maximum_Suppression Time    =       CPU(0.015839)  GPU(0.002657)    Non-maximum_Suppression + 10(0.002657)
Hysteresis_Thresholding Time    =       CPU(0.025210)  GPU(0.001277)    Hysteresis Thresholding + 10(0.001277)
total_score is 50
execution time is 0.010513
```

Non_maximum_Suppression 커널을 수행하기 전에 블록 안의 스레드 개수를 1000개로 늘렸다. 이 경우에 Non-maximum_Suppression 수행 시간이 0.01초가 늘어났다. 아무래도 min, max값을 비교할 때, 블록 안의 스레드 개수만큼 비교해야 하는데, 개수가 늘어나서 반복문이 늘어났기 때문인 것 같다.

C. Noise_Reduction과 Intensity_Gradient에서 글로벌 메모리만 사용할 경우

```
Gray_Scale Time                 =       CPU(0.007664)  GPU(0.001466)    Gray_Scale + 10(0.001466)
Noise_Reduction Time            =       CPU(0.101253)  GPU(0.001186)    Noise_Reduction + 10(0.001186)
Intensity_Gradient Time         =       CPU(0.104555)  GPU(0.003896)    Intensity_Gradient + 10(0.003896)
Non-maximum_Suppression Time    =       CPU(0.018568)  GPU(0.001268)    Non-maximum_Suppression + 10(0.001268)
Hysteresis_Thresholding Time    =       CPU(0.029441)  GPU(0.001316)    Hysteresis Thresholding + 10(0.001316)
total_score is 50
execution time is 0.009132
```

엄청 차이가 나는 건 아닌데, 그래도 조금 시간이 늘어났다.

6. Conclusion

1. Shared Memory사용

   Noise_Reduction과 Gradient_Intensity 커널 함수를 구현할 때, Shared Memory와 Global Memory에 접근할 경우를 분리해서 생각해야 했다. 이것이 수업 때도 배운 것인데 무척 헷갈렸다. 그래도 강의 교안을 보면서 구현했다. 일단 Shared Memory로만 해결할 수 있는 부분을 나누고, 그 다음 Global Memory에 접근을 할 수 밖에 없는 경우를 나눈다. 이렇게 어떤 문제를 나누면서 생각해보니 잘 구현이 되었다.

2. CPU_Func.cu 코드의 이해

   CPU_Func.cu의 코드를 보고 GPU_Func.cu에 변형하여 작성해야 했다. 반복문이 써진 곳은 전부 커널로 보내자는 생각을 가지고 접근했다. 그럼에도 불구하고 어느정도 이해가 있어야 수행시간을 줄일 수 있을 것 같았다. 그래서 CPU_Func.cu의 흐름을 파악하고 이해하려 노력했다.

3. CudaMemcpy() 함수

   CudaMemcpy() 함수도 시간을 굉장히 많이 잡아먹는 것을 알았다. 아무래도 글로벌 메모리에 접근하는 것이다 보니 시간이 걸리는 것 같다. CudaMemcpy()를 최대한 줄이니 시간이 줄었다. 메모리 관련 함수나 접근은 최소화하는 것이 좋다는 것을 배웠다.

4. constant memory 사용

   수업시간에 교수님께서 필터를 constant memory에 올려서 사용한다고 하셨다. 실제로 과제를 수행하면서 constant memory에 필터를 올리고 사용했다. 이렇게 접근하니 정말 메모리 접근 시간을 감소하는 것 같아서 뿌듯했다.