고급소프트웨어실습
기말 리포트
(CSE 4152)

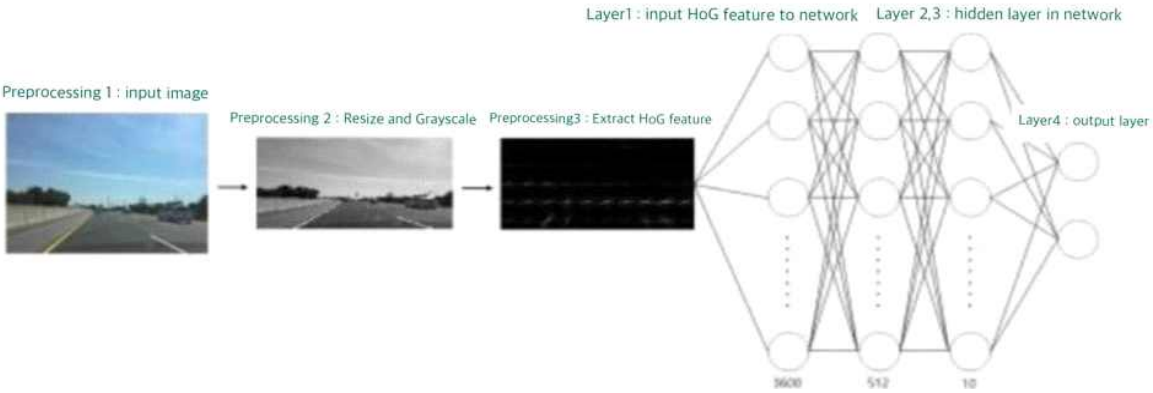
Due: 2020년, 12/17일, 오후 12시 (정오)

학번 : 20181593
반 번호 : 2반
이름 : 계인혜

1. 영상 처리 분야에서 Deep Neural Network (DNN) 기술을 사용하여 상용화된 SW 또는 제품의 사례가 있는지 조사하고, 있다면 대표적인 한 가지에 사용된 DNN 기술에 대하여 구조도, 기능, 특징을 포함하여 한 페이지 이내로 설명하시오. 만약 상용화 단계의 사례가 없다면, 연구 개발 등에 많이 활용되는 대표적인 영상 처리 DNN 기술에 대하여 같은 방식으로 설명하시오.

DNN과 HoG Feature를 이용하여 도로 소실점을 검출하는 데에 DNN 기술을 사용할 수 있다. 소실점이란 실제 공간의 평행한 선들이 영상 내에 투영되면서 한 곳에 모이는 점이다.또한 HoG는 이전 실습에서 다뤄본 적이 있는 영상 feature로, 대상 영역을 일정 크기의 셀로 분할하고, 각 셀마다 gradient magnitude가 일정 값 이상인 픽셀들의 방향에 대한 히스토그램을 구한 후 이 히스토그램의 bin 값들을 일렬로 연결한 벡터이다. 즉, HoG는 edge의 방향 히스토그램 템플릿이라고 볼 수 있다.

다음 그림은 해당 기술에 사용된 DNN의 구조도이다. 전처리 과정에서 영상을 입력받고, 영상을 resize(160*80)한 후 grayscale로 변환한다. 이후 해당 영상에서 HoG feature를 추출한다. 이를 DNN 구조에 입력하여 학습을 진행한다. 최종적으로 학습된 네트워크에 test 영상을 입력하여 layer4에서 소실점 값 좌표를(x,y) 예측한다.



최근 많은 관심을 받는 자율주행 자동차는 크게 도로 검출 기술과 이동 물체 인식 검출 기술을 필요로 한다. 이때, 도로 공간에서의 소실점의 위치를 정확히 예측할 수 있다면 도로 영역의 공간 정보들을 얻을 수 있고, 추출된 차선의 위치를 개선하거나 깊이 지도 영상을 생성할 수 있다. 1) 소실점을 사용하여 도로 검출 뿐 아니라 이동 물체도 인식할 수 있다. 따라서 DNN 기술을 사용하여 소실점을 찾는 이 방식은 자율 주행 자동차의 안전하고 정확한 이동에 큰 도움을 줄 것이다.

이 방식의 특징은 정확도 평가를 위해 Norm Distance 방법을 사용하는 것이다. 보통 두 점 사이(실제 소실점, 예측된 소실점)의 거리를 구할 때 Euclidean distance를 많이 사용하지만, 이 방법을 사용하게 되면 이미지 해상도에 따라 오차가 다르게 나올 수 있기 때문에 정확한 오차 검출을 위해 Norm distance를 사용한다. Norm distance는 다음과 같이 정의된다. 이때 diagonal은 이미지의 대각 길이를 의미한다. 즉, Norm distance는 Euclidean distance를 이미지의 대각 길이로 나누는 것이다.

$$Norm\,distance = |VPP - VPG| / diagonal$$

(VPP : 예측된 소실점의 좌표, VPG : 실제 소실점의 좌표)

윤대은, 최형일. DNN과 HoG Feature를 이용한 도로 소실점 검출 방법. 한국콘텐츠학회논문지. 2019

1) 김종환, 이충희. 관심영역 설정을 이용한 도로상 차량의 검출방법. 대구경북과학기술원. (특허출원번호 : 10-2010-0126756)

2. 아래의 (CODE 1)은 다음과 같은 C/C++ 코드를 어떤 이전 버전의 Visual Studio의 DEBUG 모드에서 컴파일하여 생성한 어셈블러 코드의 일부이다.

```
switch (val) {
    case 0: tmp = val*misc; break;
    case 1: tmp = val + misc; break;
    case 2: tmp = val/misc; break;
    /* here */
}
```

위의 C/C++ 코드의 /* here */ 부분에 다음과 같은 문장을 추가한 후 동일 조건에서 컴파일하였을 때, (CODE 2)와 같은 결과를 얻었다.

```
case 3: tmp = val - misc; break;
```

수행 속도 차원에서 어떠한 이유로 (CODE 2)의 계산 방식이 (CODE 1)의 방식보다 우월할 수 있는지, 각 방법의 작동/코드 수행 원리를 반드시 설명하면서 상세히 기술하라.

(CODE 1)

```
        mov     eax, DWORD PTR _val$[ebp]
        mov     DWORD PTR -8+[ebp], eax
        cmp     DWORD PTR -8+[ebp], 0
        je      SHORT $L544
        cmp     DWORD PTR -8+[ebp], 1
        je      SHORT $L545
        cmp     DWORD PTR -8+[ebp], 2
        je      SHORT $L546
        jmp     SHORT $L541
$L544:
        mov     ecx, DWORD PTR _val$[ebp]
        imul    ecx, DWORD PTR _misc$[ebp]
        mov     DWORD PTR _tmp$[ebp], ecx
        jmp     SHORT $L541
$L545:
        mov     edx, DWORD PTR _val$[ebp]
        add     edx, DWORD PTR _misc$[ebp]
        mov     DWORD PTR _tmp$[ebp], edx
        jmp     SHORT $L541
$L546:
        mov     eax, DWORD PTR _val$[ebp]
        cdq
        idiv    DWORD PTR _misc$[ebp]
        mov     DWORD PTR _tmp$[ebp], eax
$L541:
```

(CODE2)

```
        mov     eax, DWORD PTR _val$[ebp]
        mov     DWORD PTR -8+[ebp], eax
        cmp     DWORD PTR -8+[ebp], 3
        ja      SHORT $L543
        mov     ecx, DWORD PTR -8+[ebp]
        jmp     DWORD PTR $L568[ecx*4]
$L546:
        mov     edx, DWORD PTR _val$[ebp]
        imul    edx, DWORD PTR _misc$[ebp]
        mov     DWORD PTR _tmp$[ebp], edx
        jmp     SHORT $L543
$L547:
        mov     eax, DWORD PTR _val$[ebp]
        add     eax, DWORD PTR _misc$[ebp]
        mov     DWORD PTR _tmp$[ebp], eax
        jmp     SHORT $L543
$L548:
        mov     eax, DWORD PTR _val$[ebp]
        cdq
        idiv    DWORD PTR _misc$[ebp]
        mov     DWORD PTR _tmp$[ebp], eax
        jmp     SHORT $L543
$L549:
        mov     ecx, DWORD PTR _val$[ebp]
        sub     ecx, DWORD PTR _misc$[ebp]
        mov     DWORD PTR _tmp$[ebp], ecx
$L543:
        ...
$L568:
        DD      $L546
        DD      $L547
        DD      $L548
        DD      $L549
```

먼저 두 코드의 차이점을 살펴보고 code2가 code1 보다 우수한 이유를 살펴보도록 하자. code1의 경우 val의 값을 eax 레지스터에 저장하고, 이를 [ebp]-8로 옮긴다. 이후는 switch 문에 해당하는 부분으로 PTR [ebp]-8의 값과 case label을 각각 비교하여 해당하는 분기문으로 점프하는 구조이다. 이 과정에서 코드를 보면 알 수 있듯 code1은 case를 하나씩 비교해가며 해당하는 부분으로 점프한다. 만약 val의 값이 case label에 없는 값이라면 $L541로 점프하여 프로그램을 마무리한다. 비교를 여러 번 해서 해당하는 부분으로 점프하기 때문에 코드의 최적화가 이뤄졌다고 볼 수 없다. 예를 들어, val = 2라면, code1에서는 cmp를 3번 해야 한다. 또한, 만약 val = 5와 같은 case 구문에 해당하지 않는 경우에 불필요하게 cmp를 3번 해야 해서 비효율적이라는 것을 알 수 있다.

code2의 경우 역시 처음에는 code1과 마찬가지로 val의 값을 eax 레지스터에 저장하고, 이를 [ebp]-8로 옮긴다. 그러나 code2는 PTR [ebp]-8의 값을 하나씩 0, 1, 2와 비교하지 않고, default로 처리되는 3이상의 값을 한번에 $L543으로 점프하여 해결한다. 즉, code1과는 다르게 val = 5와 같은 case 구문에 해당하지 않는 경우에도 cmp를 1번만 하여 switch 구문을 벗어날 수 있다는 것이다.

또한, code2는 jump table을 사용하여 코드 최적화를 한다. jump table이란 swtich case label 개수만큼의 원소를 가지는 일종의 배열로 각 원소의 값에는 해당 라벨의 주소가 저장되어 있는 구조이다. 즉 라벨에 해당하는 case 문으로 가기 위해 여러 번 비교를 하지 않고, jump table을 이용하여 한 번에 갈 수 있다는 것이다. code2에서 jump table을 이용하여 해당하는 case로 바로 점프하는 코드는 mov ecx, DWORD PTR –8 +[ebp], jmp DWORD PTR &L568[ecx*4]이다. 이때 ecx에 4를 곱하는 이유는 jump table의 offset이 4 byte에 해당하기 때문이다. case가 많아질수록 code2가 code1보다 계산 방식이 유리해질 것이라고 예측해볼 수 있다.

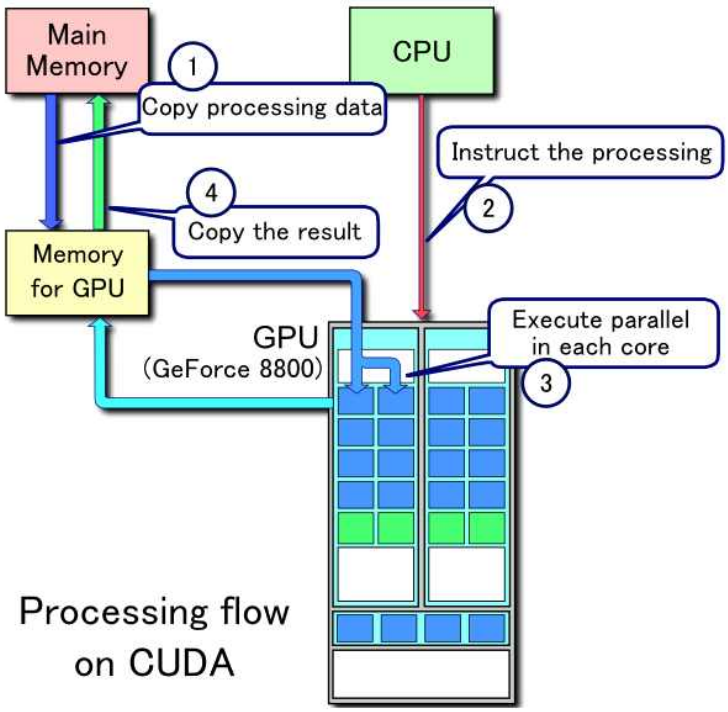
코드를 이어서 분석해 보도록 하자. 다음은 각 연산에 해당하는 부분의 라벨과 코드 설명을 정리한 표이다.

	code1	**code2**
*	val의 값을 ecx에 저장하고, misc의 값을 ecx에 곱한다. 이후 ecx의 값을 tmp에 옮긴 다음 $L541로 점프한다. (break문에 해당)	val의 값을 edx에 저장하고, misc의 값을 edx에 곱한다. 이후 edx의 값을 tmp에 옮긴 다음 $L543로 점프한다. (break문에 해당)
+	val의 값을 edx에 저장하고, misc의 값을 edx에 더한다. 이후 edx의 값을 tmp에 옮긴 다음 $L541로 점프한다. (break문에 해당)	val의 값을 eax에 저장하고, misc의 값을 eax에 더한다. 이후 eax의 값을 tmp에 옮긴 다음 $L543로 점프한다. (break문에 해당)
/	val의 값을 eax에 저장하고, misc의 값으로 eax를 나눈다. 이후 eax의 값을 tmp에 옮긴다. 마지막 분기문이기 때문에 jmp 할 필요는 없다. 이때 cdq를 사용하여 부호가 있는 비트 정보를 확장한다.	val의 값을 eax에 저장하고, misc의 값으로 eax를 나눈다. 이후 eax의 값을 tmp에 옮긴 다음 $L543로 점프한다.(break문에 해당) 이때 cdq를 사용하여 부호가 있는 비트 정보를 확장한다.
-		val의 값을 ecx에 저장하고, misc의 값을 ecx에서 뺀다. 이후 ecx의 값을 tmp에 옮긴다. 마지막 분기문이기 때문에 jmp 할 필요는 없다.

3.

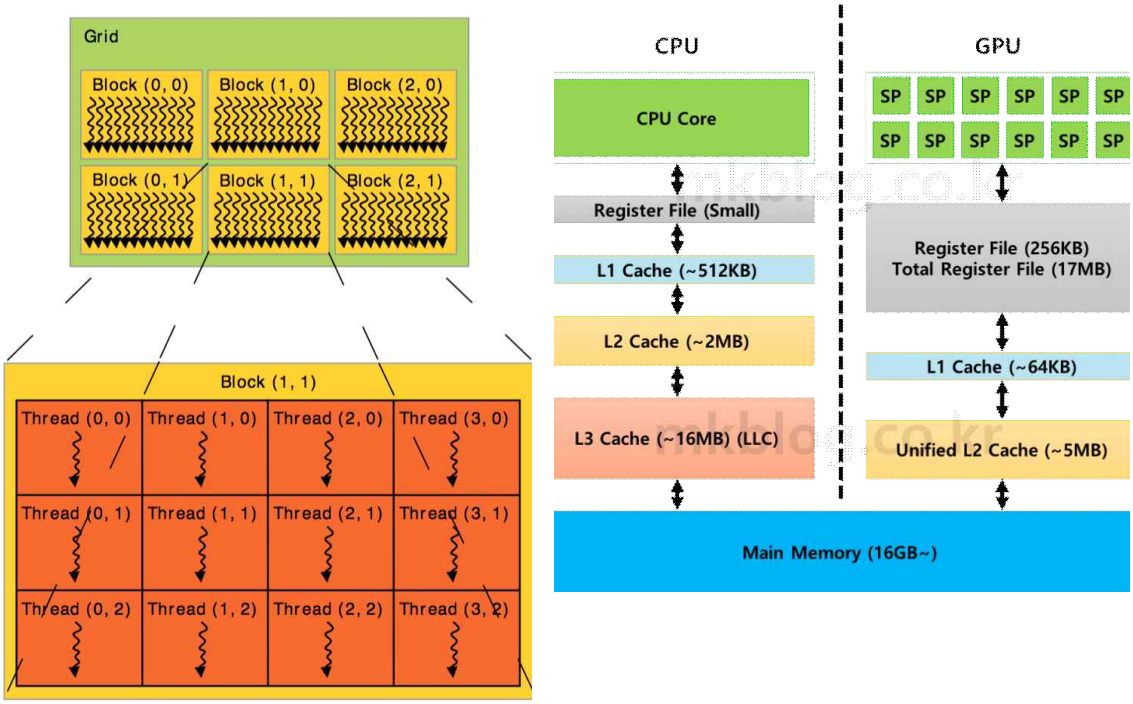
아래한글 기준 1500글자(공백 제외) 이상의 분량으로 명료하게 기술하라. 강의자료의 내용을 copy-and-paste할 경우 답으로 인정하지 않으며, 반드시 자신의 언어로 CUDA에 대하여 잘 모르는 컴퓨터공학 전공 3학년에게 설명하는 수준에서 기술할 것. (제출 내용에 대하여 copy-check를 진행할 예정임)

CUDA(Computed Unified Device Architecture)는 GPU를 이용한 개발 도구로, CUDA 사용을 위해서는 CUDA 메모리를 할당 및 해제, CUDA 메모리에 데이터 복사 그리고 for문을 풀어 cuda thread로 따로 계산하는 것이 필요하다. 이때 CUDA 함수는 __global__을 붙여 표현한다. CUDA 코어의 동작 순서를 나타낸 다음 그림을 살펴보자. 1번 과정에서는 GPU 에서 사용할 메모리를 할당하고, 데이터를 준비한다. 이 과정에서 cudaMalloc, cudaMemcpy라는 함수를 사용하게 된다. 데이터를 복사할 때 CPU에서 GPU로 복사한다면 cudaMemcpyHostToDevice을 사용하고, 반대의 경우는 cudaMemcpyDeviceToHost를 사용한다. 이때 Host는 CPU, Device는 GPU, kernel은 GPU에서 수행하는 함수를 의미한다. 2번 과정에서 CUDA 스케줄러에게 코드 실행에 대한 정보와 데이터의 크기 등을 알려줌으로써 CUDA가 multi-thread를 구동하는데 필요한 정보들을 전달한다. 다음으로는 GPU에서 각 코어에 해당하는 부분에 대한 연산(kernel)을 병렬적으로 처리한다.(3번 과정) 마지막으로 4번 과정에서 연산 결과를 메인 메모리에 저장한다.



GPU 연산 방식에는 SIMD(Single Instruction Multiple Data), MIMD, SIMT(Single Instruction Multiple Thread)가 있다. 이중 SIMT는 하나의 명령어로 여러 개의 스레드를 동작시키는 것으로 동시에 여러 개의 데이터를 병렬적으로 처리할 수 있다는 것이다. SIMD는 하나의 명령으로 여러 개의 데이터를 가공하는 계산 방식이다.

한편, CUDA의 thread hierarchy는 아래 그림과 같다. 하나의 멀티 프로세서에서 작동되는 1~3차원의 thread의 모임이 block, 1~3차원으로 block을 여러 개 묶은 것이 grid라고 정의된다. 또한 그림에는 나와 있지 않지만 warp는 CUDA에서 동시에 처리 가능한 스레드의 개수로 warp 내의 스레드들은 SIMT 방식으로 처리되고 각 warp는 독립적으로 프로세싱 된다. 이들을 적절히 인덱싱하여 각각을 구분하고 그에 해당하는 데이터를 병렬적으로 처리한다. CUDA 인덱싱 관련 내장 변수를 간단히 정리해보자. gridDim이란 kernel의 block수로 grid의 차원을 표시하며, grid의 크기를 구하는데 사용된다. blockDim이란 CUDA block의 크기를 의미하며, blockIdx는 CUDA block의 인덱스를 가지고 있다. 마지막으로 threadIdx는 블록 내에서 cuda thread의 인덱스를 의미한다.



마지막으로 GPU의 메모리 구조에 대해서 살펴보도록 하자. 위의 그림에서 CPU와 GPU의 대략적인 메모리 구조를 살펴볼 수 있다. GPU는 CPU에는 없는 shared memory를 가진다. shared memory는 블록 내의 모든 스레드가 사용가능하며, L1 캐시로 일부를 변경 가능하다. shared memory를 사용하면 필요한 만큼의 데이터만 읽어올 수 있으며, global memory 접근 시간을 줄일 수 있다. shared memory를 사용할 때는 반드시 동기화를 시켜주어야 한다. shared memory는 global 데이터를 cache처럼 shared에 올린 뒤 사용하는 것인데 각 스레드가 shared memory에 올리는 작업을 수행하기 때문에 동기화를 하지 않으면 모두 메모리에 올라오지 않은 상태에서 garbage 값을 가져오는 경우가 생기기 때문이다. 동기화를 위한 cuda 함수는 __syncthreads이다.

(그림, 공백 제외 글자 수 : 1477자)