**수치 컴퓨팅 및 GPU 프로그래밍**

**hw2 report**

20191571 김세영

**1. 환경 명세**

Operation System: Microsoft Windows 11 Education 64-bit

Compiler: Microsoft Visual Studio Community 2019

CPU: Intel(R) Core(TM) i5-10400F CPU @ 2.90GHz 2.90 GHz

GPU: NVIDIA GeForce GTX 1660 SUPER

OpenCL Version: 3.0

**2. Sobel operator 구현 방식 및 구현 내용**

Structure of Arrays(SoA)를 선택하였다.

sobel operator의 최적화 전 커널과 최적화한 커널을 구현하여 sobel 연산을 수행하고, 해당 커널 수행에 걸린 시간을 콘솔과 텍스트 파일에 출력한다.

**3. Kernel-Naive**

hw1에서 구현한 코드에서 sobel\_x, sobel\_y의 filter size를 25로 수정하였다. 또한 각 색상이 0~255 사이의 값을 갖는다고 가정할 때 한 픽셀의 sobel operator의 결과가 최대 255가 나올 수 있도록 sobel\_x, sobel\_y를 normalization 하였다.

각 픽셀의 rgb값을 grayscale로 계산하고, 그 값을 sobel operator의 필터 값과 곱해서 계산한다.

**4. Kernel-Optimized 최적화 방법**

1) loop unrolling : sobel operator는 필터 값 sobel\_x와 sobel\_y을 grayscale로 변환한 픽셀값과 각각 곱하면서 계산한다. 이 과정에서 sobel\_x와 곱한 값은 int g\_x, sobel\_y와 곱한 값은 int g\_x에 더해지도록 하였다. 이때 이중 for loop(25번 반복)의 안쪽에 있는 for loop을 풀어서(5번 반복) loop unrolling을 적용하였다.

2) local memory : \_\_local uchar\* shared\_mem은 local memory로, input\_data의 r, g, b값을 읽어들여 grayscale로 변환한 후 저장한다. (매크로 GRAYSCALE 정의하여 사용) grayscale에서는 r, g, b 값이 같기 때문에 uchar3\* 가 아닌 uchar\* type으로 저장한다. local memory는 work-group단위로 할당되기 때문에 shared\_mem에 grayscale 값을 저장하는 작업을 수행한 후 output\_data에 저장하기 전에 barrier()를 호출하여 synchronization을 한다. 각 work-group 안에 있는 work-item끼리는 shared\_mem의 데이터를 공유하여 사용할 수 있다.

이 방법을 통해 필터 값을 픽셀의 rgb 값과 곱할 때 input data의 rgb, 즉 input data(global memory)에 3번 접근해야 했던 방식을 미리 계산해둔 shared\_mem의 grayscale 값을 접근하는 것으로 바꾸어 global memory access를 줄일 수 있다.



Kernel\_Naive



Kernel\_Optimized

**5. configuration 사용법**

Source/Config\_SoA\_AoS.h에서 확인할 수 있다.

1 )IMAGE\_OPERATION: 수행할 kernel을 지정할 수 있다.

SoA\_NAIVE Kernel\_Naive를 수행한다.

SoA\_OPTIMIZED Kernel\_Optimized를 수행한다.

2) INPUT\_IMAGE : input image를 지정할 수 있다. 0부터 10까지의 정수가 들어올 수 있다.

3) STAT\_FILE\_NAME: GPU 커널 수행에 걸린 시간(QUEUED to END)을 측정하여 txt 파일로 저장한다. 이 때 저장할 파일 명을 지정할 수 있다.

4) N\_EXECUTIONS 커널 반복 수행 횟수. 시간을 측정할 때는 평균을 내어 구한다.

5) LOCAL\_WORK\_SIZE\_N: 해당 프로그램은 work-group을 2차원으로 가정한다.

LOCAL\_WORK\_SIZE\_0,1은 각각 work group의 dim 0, dim 1의 size이다.

6) INPUT(OUTPUT)\_FILE\_NAME: 읽을/쓸 이미지 파일명을 지정한다.

**6. 실험 결과 (단위: ms)**

N\_EXECUTIONS: 20, Speed up: SoA\_NAIVE/SoA\_OPTIMIZED

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| INPUT\_IMAGE : 0 (size: 7360x4832) | | | | | |
| Work group size | (8,16) | (16,16) | (32,16) | (64,16) | (32,32) |
| SoA\_NAÏVE | 16.07 | 14.726 | 16.476 | 19.778 | 20.024 |
| SoA\_OPTIMIZED | 9.582 | 10.686 | 13.467 | 16.802 | 16.409 |
| Speed up | 1.677103 | 1.378065 | 1.223435 | 1.177122 | 1.220306 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| INPUT\_IMAGE : 5 (size: 1856x1376) | | | | | |
| Work group size | (8,16) | (16,16) | (32,16) | (64,16) | (32,32) |
| SoA\_NAÏVE | 1.4 | 1.29 | 1.454 | 1.793 | 1.873 |
| SoA\_OPTIMIZED | 0.826 | 0.875 | 1.137 | 1.456 | 1.522 |
| Speed up | 1.694915 | 1.474286 | 1.278804 | 1.231456 | 1.230618 |

**7. 결과 분석**

실험의 모든 결과가 SoA\_OPTIMIZED가 SoA\_NAIVE보다 빠르게 측정되었다. SoA\_OPTIMIZED는 Loop unrolling과 local memory를 적용했기 때문에 SoA\_NAIVE보다 branch가 적게 생성되고, global memory access 횟수가 적기 때문에 수행시간이 줄어든 것이다.

input image의 크기가 큰 경우(INPUT\_IMAGE: 0)와 크기가 작은 경우(INPUT\_IMAGE:5) 두가지로 나누어 실험을 진행하였다. 두 경우의 Speed up은 비슷한 경향을 보였다. 실험한 모든 workgroup size에서 image 크기가 작은 경우(INPUT\_IMAGE:5)의 Speed up이 image 크기가 큰 경우(INPUT\_IMAGE: 0)보다 높았다.

SoA\_NAIVE와 SoA\_OPTIMIZED가 가장 빠르게 실행된 workgroup size는 (16,16)이고, 가장 느리게 실행된 workgroup size는 (32,32)이다. input image가 0일 때의 SoA\_OPTIMIZED는 (64,16)에서 가장 느리게 실행되었다.

커널을 실행했을 때 CL\_KERNER\_WORK\_GROUP\_SIZE는 256인 것을 콘솔에서 확인할 수 있다.



\*naive와 optimized 동일

(32.,32)와 (64,16)의 workgroup size는 256을 넘기 때문에 느리게 실행되고, (16,16)은 workgroup size가 256이기 때문에 빠르게 실행된다고 추측할 수 있다.

(64,16)과 (32,32)의 workgroup size는 같지만, (64,16)일 때의 locality가 (32,32)의 경우보다 높기 때문에 cache miss가 덜 발생할 것이다. 따라서 input image 0의 optimized를 제외한 3가지 경우에서 (64,16)의 수행시간이 더 적게 걸린다.