**수치 컴퓨팅 및 GPU 프로그래밍**

**hw3 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. Configuration**

Source/config\_ConcurrentCopyCompute.h에서 확인할 수 있다.

1) MAXIMUM\_COMMAND\_QUEUES: command queue의 최대 개수. 2의 n제곱으로 설정해야한다.

2) N\_KERNEL\_LOOP\_ITERATIONS: kernel을 수행할 loop 횟수. kernel 수행 시 같은 코드를 K\_KERNEL\_LOOP\_ITERATIONS 만큼 수행한다.

3) N\_KERNEL\_CALL\_ITERATIONS: kernel을 호출할 횟수.

4) LOCAL\_WORK\_SIZE\_0, LOCAL\_WORK\_SIZE\_1: local work size, 각각 dim 0(x), 1(y)이다. 이 때 LOCAL\_WORK\_SIZE\_1은 input image의 height를 segment(multiple queue의 경우에는 command queue)의 개수로 나눈 값에 나누어 떨어져야 한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

5) INPUT\_IMAGE: 입력으로 사용할 영상을 선택한다. 0,1,2,3,4,5,8,9,10 이 들어올 수 있다.

6) COPY\_COMPUTE\_TYPE: multiple queues와 three queues 방법 중 하나를 선택한다. COPY\_COMPUTE\_TYPE\_THREE\_QUEUES\_WITH\_EVENTS, COPY\_COMPUTE\_TYPE\_MULTIPLE\_QUEUES 중 하나가 들어올 수 있다.

7) COPY\_COMPUTE\_COMMANDS\_PROFILING: NO\_437, YES\_437 중 하나를 선택할 수 있고, YES\_437 를 선택한 경우에는 각 queue에서 data transfer from host to device, compute, data transfer from device to host의 걸린 시간, 시작 시점과 종료 시점이 콘솔상에 출력된다.

**3. 구현 내용**

**1) segment 분할 방법**

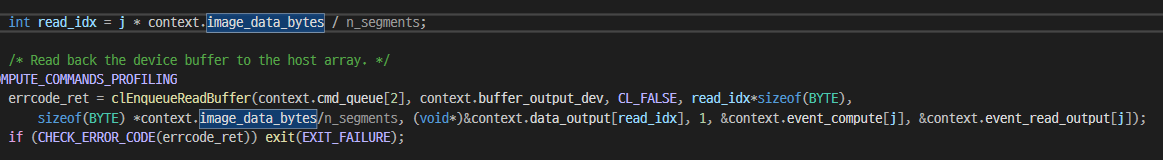
각 segment의 width는 context.image\_width(입력 영상의 width)와 동일하며, height는 입력 영상의 height를 segment의 개수로 나눈 것이다. sobel operator의 filter size가 5x5이기 때문에 맨 앞과 맨 뒤 segment의 height는 context.local\_work\_size[1]만큼, 그렇지 않은 segment의 height는 context.local\_work\_size[1]\*2 만큼 더한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

clEnqueueWriteBuffer에서 써야하는 buffer의 크기를 size, input\_offset을 start\_idx에 저장한다.

clEnqueueReadBuffer에서는 context.image\_data\_bytes를 segment의 크기로 나눈 것 만큼 읽는다.



**2) Kernel\_Optimized**

hw2에서 SoA로 구현하였지만 계산과 data transfer의 편의성을 위해 AoS로 수정하여 구현하였다. 현재 segment가 마지막 segment인지 확인하기 위해 parameter로 uchar last\_flag를 추가하였다.

기존 커널 코드에서 가장자리 작업을 수행하는 process\_boundary\_work\_groups\_AoS에서 output\_data에 계산한 결과값을 assign하는 부분을 수정했는데, 1)에서 clEnqueueWriteBuffer로 쓴 size와 clEnqueueReadBuffer에서 읽는 size가 다르기 때문에 실제로 clEnqueueReadBuffer에서 읽을 부분에 해당하는 index만 output\_data에 계산한 결과값(intensity)을 assign한다.



**3) THREE QUEUES WITH EVENTS 구현 내용**

context.buffer\_input\_pinned는 pinned memory에 있는 buffer object이다. 이 buffer object의 mapped pointer은 context.data\_input이기 때문에 함수 호출 전 context.AoS\_image\_input에 저장된 데이터를 context.data\_input으로 copy한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

영상을 개의 segment로 나누어서 context.cmd\_queue[0]에서 data transfer from host to device, context.cmd\_queue[1]에서 compute, context.cmd\_queue[2]에서 data transfer from device to host를 수행한다. 2)에서 각 compute과정에서는 clEnqueueReadBuffer에서 읽을 부분만 값을 assign했고, 각 segment의 읽는 부분은 겹치지 않기 때문에 data transfer from device to host 사이에 다른 segment의 compute가 발생해도 읽는 값은 바뀌지 않는다.

segment의 개수에 따라 context.data\_output(data transfer from device to host)값이 다르지 않다는 것을 확인하기 위해 반복문 수행 전 use\_multiple\_segments\_and\_three\_command\_queues\_with\_events\_breadth(1) 를 수행한 값을 context.solution에 저장한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

각 반복문 수행마다 check\_correctness\_on\_host()를 호출하고, 이 함수에서는 주어진 util 함수를 수정한 util\_compair\_two\_byte\_arrays(context.solution, context.data\_output, context.image\_data\_bytes, 0)를 호출하여 context.solution과 context.data\_output의 값을 비교한다.

텍스트, 화면이(가) 표시된 사진

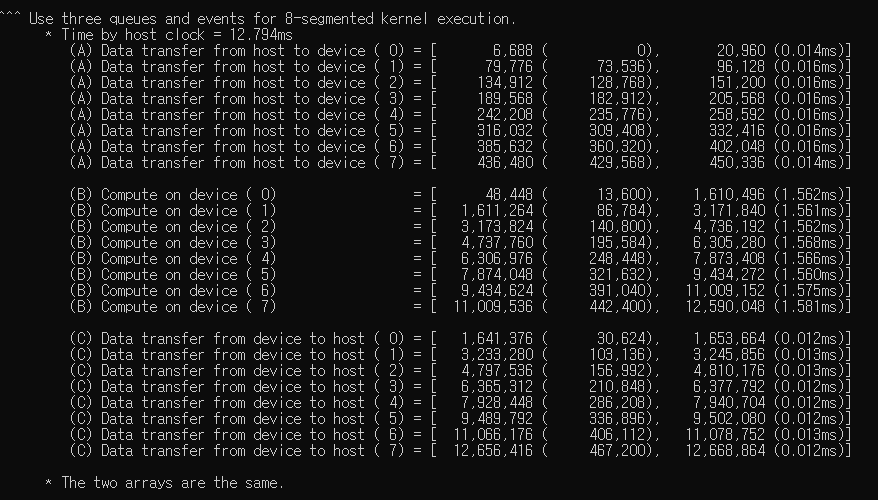
자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

segment 개수가 1,2,4,8,16,32개 일 때의 계산 결과는 모두 동일하다.

segment 개수가 8개 일 때의 profiling을 살펴보면, segment마다 data transfer from host to device가 종료된 이후에 compute가 실행되고, compute가 종료된 이후에 data transfer from device to host가 실행되는 것을 확인할 수 있다. 각 segment마다 data transfer from host to device, compute, data transfer from device to host는 dependency가 설정되어 있다. 또한, context.cmd\_queue는 모두 in-order command-queue이기 때문에 이전 segment의 data tranfer(compute)가 종료된 이후에 다음 segment의 data transfer(compute)가 수행된다.



**4) MULTIPLE QUEUES 구현 내용**

segment 개수 만큼의 command queue가 생성되고, 한 segment의 data transfer from host to device, compute, data transfer from device to host가 하나의 command queue에서 이루어진다.

텍스트이(가) 표시된 사진

자동 생성된 설명

코드 내의 context.cmd\_queue의 index는 j로 모두 같다.

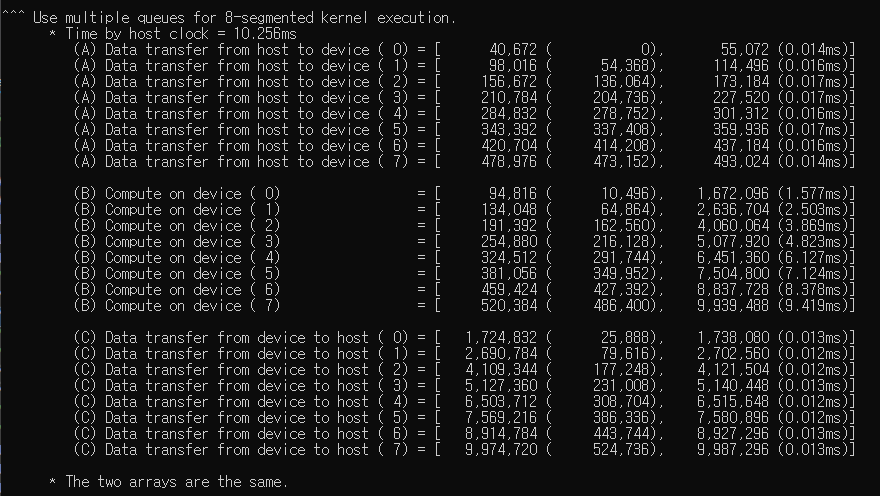
텍스트이(가) 표시된 사진

자동 생성된 설명

segment 개수가 1,2,4,8,16,32개 일 때의 계산 결과는 모두 동일하다. 이 수행 결과의 configuration은 3)과 동일하다.

segment 개수가 8개 일 때의 profiling을 살펴보면, segment마다 data transfer from host to device가 종료된 이후에 compute가 실행되고, compute가 종료된 이후에 data transfer from device to host가 실행되는 것을 확인할 수 있다. 각 segment마다 data transfer from host to device, compute, data transfer from device to host는 dependency가 설정되어 있다. 또한, 이전 segment의 data tranfer(compute)가 종료된 이후에 다음 segment의 data transfer(compute)가 수행된다.

명시적으로 dependency를 선언하지 않은 경우 이전 compute가 끝나기 전에 다음 compute가 시작되는 경우가 있어서 COPY\_COMPUTE\_COMMANDS\_PROFILING 가 YES\_437인 경우 명시적으로 dependency를 선언하였다.



선언전 profiling

텍스트이(가) 표시된 사진

자동 생성된 설명

선언 후 profiling, 수행시간이 늘어났다. 다만 dependency 선언 여부와 상관없이 data\_output은 항상 같기 때문에 시간 측정시에는 (NO\_437) dependency를 명시적으로 선언하지 않았다.

**4. 실험 결과**

2.configuration에서 언급했듯이 LOCAL\_WORK\_SIZE\_1은 input image의 height를 segment(multiple queue의 경우에는 command queue)의 개수로 나눈 값에 나누어 떨어져야 하기 때문에 image의 width와 height가 2의 n제곱인 INPUT\_IMAGE 8(2048x2048),10(1024x1024)를 이용하여 실험을 진행하였다.

텍스트, 오렌지이(가) 표시된 사진

자동 생성된 설명



측정한 시간은 Host에서의 수행 시간이다. (단위:ms)

1. INPUT\_IMAGE:8

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| three queues | INPUT\_IMAGE : 8 (size: 2048x2048) | | | |
| n\_segment work\_group\_size | (16,16) | (32,16) | (64,16) | (32,32) |
| 1 | 115.774 | 134.712 | 150.859 | 151.535 |
| 2 | 114.912 | 133.648 | 150.2 | 152.187 |
| 4 | 115.401 | 133.904 | 149.913 | 152.295 |
| 8 | 116.473 | 134.676 | 152.929 | 156.307 |
| 16 | 118.479 | 138.723 | 155.499 | 160.309 |
| 32 | 124.485 | 142.532 | 157.836 | 170.939 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| multiple queues | INPUT\_IMAGE : 8 (size: 2048x2048) | | | |
| n\_segment work\_group\_size | (16,16) | (32,16) | (64,16) | (32,32) |
| 1 | 116.436 | 133.919 | 149.83 | 152.517 |
| 2 | 115.46 | 133.483 | 149.71 | 152.053 |
| 4 | 115.243 | 133.291 | 149.292 | 152.382 |
| 8 | 115.835 | 133.92 | 150.044 | 154.704 |
| 16 | 117.745 | 135.959 | 151.747 | 159.404 |
| 32 | 121.752 | 140.896 | 155.989 | 169.557 |

1. INPUT\_IMAGE : 10

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| three queues | INPUT\_IMAGE : 10 (size: 1024x1024) | | | |
| n\_segment work\_group\_size | (16,16) | (32,16) | (64,16) | (32,32) |
| 1 | 29.569 | 33.85 | 37.285 | 37.801 |
| 2 | 29.007 | 33.661 | 37.661 | 38.375 |
| 4 | 29.394 | 34.291 | 37.774 | 38.997 |
| 8 | 30.206 | 35.075 | 38.104 | 39.924 |
| 16 | 32.066 | 36.509 | 38.958 | 43.108 |
| 32 | 35.185 | 39.27 | 53.066 | 55.342 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| multiple queues | INPUT\_IMAGE : 10 (size: 1024x1024) | | | |
| n\_segment work\_group\_size | (16,16) | (32,16) | (64,16) | (32,32) |
| 1 | 28.989 | 33.782 | 37.613 | 38.673 |
| 2 | 28.715 | 33.502 | 37.368 | 38.244 |
| 4 | 28.906 | 33.554 | 37.33 | 38.452 |
| 8 | 29.268 | 33.918 | 37.561 | 39.548 |
| 16 | 30.169 | 34.782 | 38.519 | 42.164 |
| 32 | 31.954 | 36.723 | 40.804 | 48.716 |

**5. 결과 분석**

1) multiple queues와 three queues with events 모두 두 영상에서 work group size (16,16)일 때 가장 수행시간이 적게 걸렸고, work group size (32,32)일 때 수행시간이 가장 오래 걸렸다. segment 개수가 1개이고 1개의 queue에서 실행된 hw2에서는 영상의 크기가 클 때는 (7360x4832) work group size (64,16)일 때 수행시간이 가장 오래 걸렸고, 영상 크기가 크지 않을 때는 (1856x1376) work group size(32,32)에서 수행시간이 가장 오래 걸렸다. 두 영상 모두 (16,16)에서 가장 수행시간이 적게 걸렸다. 본 실험(hw3)에서 사용된 입력 영상의 크기는 2048x2048, 1024x1024이므로 hw2의 영상 크기가 크지 않았을 때(1856x1376)의 실험 결과의 경향성을 따라간다고 할 수 있다.

테이블이(가) 표시된 사진

자동 생성된 설명 hw2에서의 실험 결과

2) three queues with event는 input image 8, work group size (64,16)을 제외하고 segment 개수가 2개일 때 가장 수행시간이 적게 걸렸고, input image 8, work group size (64,16)일 때는 segment 개수가 4개일 때 가장 수행시간이 적게 걸렸다. segment 개수가 32개일 때 수행시간이 가장 오래 걸렸다. segment 개수가 1보다 크면 한 segment가 compute를 수행하는 동안 다른 segment의 data transfer를 할 수 있기 때문에 수행시간이 줄어들 수 있다. 따라서 segment개수가 2개(input image 8, work group size (64,16)일 때는 4개 )일 때 수행시간이 가장 적게 걸린 것이다. 단, segment 개수가 많으면 각 segment마다 boundary에 해당하는 데이터를 추가로 보내기 때문에 clEnqueueWriteBuffer에서 host에서 device로 보내는 데이터 양이 늘어나고, 계산해야 할 양도 늘어나기 때문에 segment 개수가 32개 일 때 수행시간이 가장 오래 걸렸다.

3) multiple queues에서는 input image가 8일 때 work group size (32,32)를 제외하고 segment개수가 4개일 때의 수행시간이 가장 적게 걸렸다. input image가 8이고 work group size가 (32,32)일 때는 segment 개수가 2개일 때 수행시간이 가장 적다.

input image가 10일때는 work group size (64,16)를 제외하고 segment개수가 2개일 때의 수행시간이 가장 적게 걸렸다. input image가 10, work group size (64,16)일 때는 segment 개수가 4개일 때의 수행시간이 제일 적다.

두 영상 모두 segment 개수가 32개일 때 수행시간이 가장 오래 걸렸다. segment 개수가 2나 4일 때 수행 시간이 가장 적게 걸리는 이유와 32일 때 수행시간이 가장 오래 걸리는 이유는 2) 와 같다.

4) multiple queues와 three queues를 비교했을 때 대체적으로 multiple queues가 더 빠르다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| three-multiple | INPUT\_IMAGE : 8 (size: 2048x2048) | | | |
| n\_segment work\_group\_size | (16,16) | (32,16) | (64,16) | (32,32) |
| 1 | -0.662 | 0.793 | 1.029 | -0.982 |
| 2 | -0.548 | 0.165 | 0.49 | 0.134 |
| 4 | 0.158 | 0.613 | 0.621 | -0.087 |
| 8 | 0.638 | 0.756 | 2.885 | 1.603 |
| 16 | 0.734 | 2.764 | 3.752 | 0.905 |
| 32 | 2.733 | 1.636 | 1.847 | 1.382 |

아래의 표는 three queues의 수행시간에 multiple queues를 뺀 값이다. segment 개수가 클수록 대체적으로 multiple queues의 속도가 더 빠른 것을 확인할 수 있다.

multiple queues는 segment개수 만큼 command queue를 사용하기 때문에 segment 개수가 1,2개인 경우는 three queues 보다 command queue의 수가 적다. 따라서 segment 개수가 4개일 때까지는 three queues가 multiple queue보다 빠른 경우가 있지만, segment 개수가 8,16,32로 3개와 큰 차이가 날 때는 그만큼 동시에 수행할 수 있는 개수가 늘어나기 때문에 multiple queue의 수행속도가 더 빠르다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| three-multiple | INPUT\_IMAGE : 10 (size: 1024x1024) | | | |
| n\_segment work\_group\_size | (16,16) | (32,16) | (64,16) | (32,32) |
| 1 | 0.58 | 0.068 | -0.328 | -0.872 |
| 2 | 0.292 | 0.159 | 0.293 | 0.131 |
| 4 | 0.488 | 0.737 | 0.444 | 0.545 |
| 8 | 0.938 | 1.157 | 0.543 | 0.376 |
| 16 | 1.897 | 1.727 | 0.439 | 0.944 |
| 32 | 3.231 | 2.547 | 12.262 | 6.626 |