멀티코어 컴퓨팅 HW #5

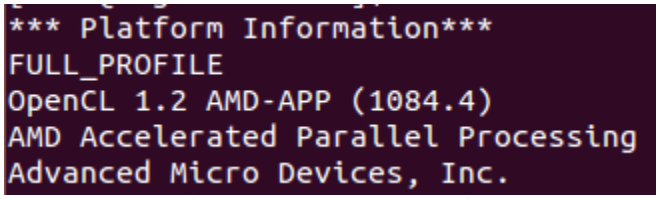
2010-11904
최재민

1. Matrix Multiplication

OpenCL 을 사용하기 위해서는 host program 과 kernel 두 종류의 코드가 필요하다. 이번 행렬 계산을 위해서는 기존 mat_mul.c 를 수정하여 host program 을 작성하였고, init.cl 과 compute.cl 이라는 두 kernel 코드를 새로 작성하여 각각 행렬 초기화 과정과 실제 행렬곱 계산을 수행하도록 하였다.

1. 1. mat_mul.c

이 코드는 host 에서 실행되는 host program 으로, OpenCL 작업을 위한 준비 과정과 kernel 들의 실행, 그리고 사용자에게 결과를 알려주는 부분을 담당하고 있다. 가장 처음에는 OpenCL 플랫폼이 존재하는지 알아보고, 플랫폼이 하나 이상 존재한다면 첫 번째 플랫폼을 사용하도록 되어 있다.



<그림 1-1. 플랫폼 정보>

코드의 가장 처음 부분에 있는 #define DEBUG 부분을 1 로 세팅 후 코드를 실행하면 플랫폼 정보를 포함한 몇몇 정보들을 위와 같이 얻을 수 있다. 위 결과를 보면 현재 선택된 플랫폼은 OpenCL 1.2 AMD-APP 을 사용하며 AMD Accelerated Parallel Processing 이라는 이름을 가지고 있다는 것을 알 수 있다. 플랫폼을 선택한 후에는 어떤 compute device 를 사용할 것인지를 선택하게 되며, 코드의 앞부분에 위치한 #define USE_GPU 에 따라 CPU 를 사용할 지 아니면 GPU 를 사용할 지를 결정하게 된다. USE_GPU 가 0 이라면 CPU 를 선택하게 되며 (여러 CPU 가 존재할 경우 첫 번째 CPU 선택), USE_GPU 가 1 이라면 GPU 를 선택하게 된다 (여러 GPU 가 존재할 때도 마찬가지). 천둥에서 실행할 경우 CPU 는 Intel Xeon E5-2650 (Sandy Bridge-EP, 8-core 2.00GHz)이며, GPU 는 AMD Radeon HD 7970 이다.

Compute device 선택이 끝나면 이에 대한 context 와 device 에 붙일 in-order command queue 를 생성하게 되며, init.cl 과 compute.cl 을 file I/O 를 통해 불러온 뒤 프로그램을 만들고 내재된 OpenCL 컴파일러를 사용하여 build 하게 된다. 여기서 한 가지 주목할 점은 kernel 을 컴파일 할 때 build option 을 줄 수 있다는 것인데, kernel 에 predefined 상수 등을 넘겨줄 때 유용하게 사용할 수 있다. 이 build option 의 필요성은 추후에 compute.cl 의 구현에 대해 설명할 때 더 자세히 이야기하도록 하겠다. 컴파일이 된 이후에는 kernel 을 실제로 만들게 되며, 이 과정까지 끝나면 argument 세팅을 제외한 kernel 실행 준비가 완료된다.

이후에는 행렬들을 저장할 메모리 공간을 host 에 할당받고, kernel 들이 계산을 수행할 수 있도록 device memory 에 올라갈 buffer 들을 생성한다. 이 때 CL_MEM_USE_HOST_PTR 라는 옵션을 사용하는데, 이는 host memory 에 저장된 행렬들을 사용하겠다는 것이며, device implementation 에 따라 caching 을 사용할 수 있도록 해 준다. 그 이후에는 kernel 의 work size 를 정해주는데, global work size 와 local work size 를 정할 수 있다. 각 work size 의 dimension 은 최대 3 차원까지 가능하며, 이번 행렬 계산에서는 2 차원 work size 를 사용한다. Global work size 행렬은 전체 work item 의 개수를 의미하고, local work size 행렬은 각 work group 의 work item 수를 의미한다. Work group 은 Nvidia 에서의 warp 와 같은 개념이며, work group 단위로 스케쥴링되며 context switch 가 일어난다. 또한 같은 work group 내에 있는 work item 들 간에만 barrier 를 이용한 synchronization 이 가능하며, synchronization 에 대해서는 compute kernel 에서 더 살펴보도록 하겠다. 한 가지 주의해야 할 점은 compute device 마다 work group size 의 최대치가 정해져 있다는 것인데, 천둥의 CPU 는 총 1024 개로 local work size 가 1024 의 제곱근인 32 를 넘을 수 없고, 천둥의 GPU 는 총 256 개로 local work size 가 256 의 제곱근인 16 을 넘을 수 없다.

두 kernel 중 먼저 실행되는 init kernel 의 경우는 각 work item 이 행렬 원소 1 개를 초기화하므로 총 NDIM * NDIM 개의 global work item 이 필요하다. (여기서 NDIM 은 행렬의 한 dimension 의 크기를 의미하며 코드 앞부분에 #define 으로 정의되어 있다) 다음에 실행되는 compute kernel 은 tiling 을 사용하기 때문에 각 work group 이 한 tile 을 담당하도록 구현되어 있고, 따라서 global work item 의 수는 행렬 원소의 수와 일치, local work item 의 수는 tile 의 크기과 일치하도록 되어 있다. 이와 같이 work size 세팅이 끝나고 kernel 에게 넘겨줄 argument 들 또한 세팅이 끝나면 kernel 을 command queue 에 넣어 실행하게 된다. 먼저 실행되는 init kernel 은 local work size 가 정의되어 있지 않고 NULL 로 주어지는데, 이럴 경우 OpenCL runtime 이 알아서 적당한 local work size 를 정해주게 된다. Init kernel 의 실행이 끝난 뒤에는 A 와 B 의 초기값들을 사용자에게 출력하여 보여줄 수 있도록 host memory 에 다시 읽어오고, compute kernel 을 enqueue 하여 실제 행렬곱 계산을 수행하게 된다.

전체 행렬곱의 수행 시간은 compute kernel 의 수행 속도에 달려 있으며, compute kernel 이 처음 enqueue 될 때 시간 측정을 시작하고, clFinish()를 통해 command queue 의 모든 kernel 들이 수행이 완료되면 compute kernel 역시 수행이 완료된 것이므로 이때 시간 측정을 끝낸다. 이후 사용자에게 수행 시간을 알려주고 앞서 할당했던 메모리 영역, 프로그램, kernel, command queue 등을 모두 해제하고 프로그램을 종료한다.

GPU 의 경우 행렬의 크기가 매우 커지면 (천둥의 GPU memory 는 3GB 로, 3 * 4 * NDIM^2 <= 3 * 2^30 에서 NDIM <= 2^14 이어야 한다) device memory 에 행렬들이 모두 올라가지 못하는 문제가 발생한다. 이를 해결하기 위해 행렬들을 일정한 정사각형 크기의 블락으로 나누어 올린 뒤 계산하는 방법을 고안하였으나, 문제는 한 블락을 올릴 때 올려야 하는 행렬들의 주소값이 연속적이지 않다는 것이었다. 이를 해결해주는 OpenCL 함수가 clEnqueueReadBufferRect 와 clEnqueueWriteBufferRect 인데, 이 함수들은 이론적으로는 사각형 영역의 메모리 공간을 읽고 쓸 수 있으나, 실제로 사용해본 결과 할당한 영역만큼을 읽고 써주는 것이 아니라 그 영역의 크기만큼의 연속적인 메모리 공간을 읽고 쓰는 버그가 있었다. 이 함수들을 사용하지 않는 경우에는 한 행렬씩 읽고 써야 하며, 행렬 크기가 클 경우 오버헤드가 상당할 것으로 예상된다. 이처럼 블락을 사용하여 큰 행렬곱을 처리하는 알고리즘은 기존의 tiling 방법과 상당히 유사하며, 대략적으로 다음과 같다. 먼저 device memory 에 행렬 3 개가 모두 올라갈 수 있을 만큼 블락의 크기를 정해준 뒤 buffer 를 생성해 준다. 또 C 행렬의 블락 하나의 부분적인 결과들을 저장할 공간을 host memory 에 할당하여 준다. C 행렬의 한 블락에 대해 필요한 A 와 B 블락들을 1 개씩 clEnqueueWriteBufferRect 를 이용하여 버퍼에 써주고, compute kernel 을 실행한다. 실행이 끝나면 C 에는 부분적인 결과가 들어있으므로, 이를 clEnqueueReadBufferRect 를 통해 아까 host memory 에 할당한 공간에 넣어준다. 이를 C 블락을 완전히 계산할 때까지 반복하여 주고, C 블락 1 개에 대해 이 과정이 끝나면 host memory 에 저장된 부분적인 결과들을 모두 더해 C 블락을 완성한다. 여기까지가 1 개의 C 블락에 대한 것이며, 이를 모든 C 블락들에 대해 반복하면 C 행렬 전체의 계산이 완료된다. 이 코드는 mat_mul_blk.c 에서 347 ~ 622 라인에서 살펴볼 수 있다.

1. 2. init.cl

이 kernel 은 가장 처음 실행되어 행렬 A 와 B 를 초기화해 주는 것을 담당한다. 코드 자체는 아주 간단하게 구성되어 있는데, 먼저 get_global_id() 함수를 통해 자신의 global work item ID 를 받아온다.
하나의 work item 이 행렬 A 와 B 의 한 행렬 원소씩을 담당하므로 이 ID 를 행렬의 index 로 사용하여 해당 원소들을 초기화한다. 여기서 초기화 값은 kernel 의 argument 로 주어지는 startNum 값에 자신의 index 를 더하여 알 수 있다.

1. 3. compute.cl

이 kernel 은 init kernel 이 실행된 뒤 실제 C = A * B 의 행렬곱을 수행하기 위해 실행된다. Naive 한 행렬곱 코드에서는 init 과 비슷하게 한 work item 이 행렬 C 의 한 원소를 계산하도록 되어 있으며, 한 work item 이 행렬 A 의 한 행 전체와 행렬 B 의 한 열 전체를 접근하므로 실행 속도가 매우 느리다. 하지만 tiling 을 구현한 이 코드를 이용하면 모든 work item 이 접근할 수 있는 global memory 가 아닌 한 work group 만 접근 가능한 local memory 에 대한 접근을 늘려 훨씬 빠른 속도를 얻을 수 있다.
OpenCL 에서는 host memory ↔ device memory ↔ local memory 에 대한 데이터 이동을 사용자가 명시적으로 정의해주어야 하며, 이 kernel 코드에서는 TA 와 TB 라는 local memory 영역을 각 work group 에 할당해 주어 각 work group 이 이 영역을 공유하며 사용할 수 있도록 해 주었다. 또한 각 work item 의 private memory 에 저장되는 acc 라는 변수를 정의해 주었는데, 이 변수에 각 work item 이 담당하는 C 행렬 원소의 값을 계속 accumulate 해주게 된다. 앞에서 build option 에 TDIM 을 넣어줘야 하

는 이유는 TA 와 TB 행렬들을 define 할 때 행렬의 크기가 필요한데 이 크기가 행렬 크기에 따라 바뀌고, OpenCL kernel 코드에서는 행렬의 크기를 변수로 주는 것이 불가능하기 때문이다. 즉, TDIM 을 kernel argument 로 넘겨주면 안되고 compile option 을 통해 상수로 넘겨줘야 하는 것이다.

실제 계산은 다음 과정에서 각 타일마다 이루어지는데, 먼저 각 타일 안에서 해당 work item 이 담당하는 위치를 찾고, 그 위치의 계산에 필요한 값들을 local memory 에 읽어오게 된다. 여기서 주의해야 할 점은 global memory 에서 local memory 로 읽어오는 작업을 work group 단위로 수행하므로 accumulation 을 진행하기 전에 work group 내의 모든 work item 들이 데이터를 읽어왔다는 것이 보장되어야 한다. 이 이유 때문에 synchronization 의 일종인 barrier 를 사용하였으며, CLK_LOCAL_MEM_FENCE 옵션은 barrier 가 끝나기 전에 local memory 에 대한 read 와 write 가 모두 완전히 실행되어야 한다는 것을 의미한다. Accumulation 다음 부분에도 barrier 가 필요한데, 다음 tile 들을 local memory 로 옮겨오기 전에 accumulation 이 끝나지 않으면 local memory 에 다른 값들이 저장되어 있을 수 있어 결과가 틀리게 나올 수 있기 때문이다. 모든 work item 들이 이 kernel 을 수행하면 C 행렬의 모든 원소들의 값이 계산되므로 kernel 실행이 끝난 뒤 host program 에서 global memory 의 C 행렬을 읽어올 수 있다.

1. 4. 실험 방법

프로그램 실행을 하기 위해 몇 가지 세팅해야 하는 것들이 있는데, 이들은 mat_mul.c 의 앞부분에 #define 으로 정의되어 있다. 먼저 DEBUG 는 플랫폼과 디바이스 정보, kernel build option 등을 알려주는 플래그로 기본값은 0 이며, 1 로 세팅하면 해당 정보들을 볼 수 있다. USE_GPU 값은 앞에서 설명하였듯이 CPU 를 사용하고 싶으면 0, GPU 를 사용하고 싶으면 1 로 세팅하면 되며, 기본값은 1 이다. NDIM 값은 행렬의 한 dimension 크기이며, 기본값은 10000 이다. TDIM 값은 한 타일의 dimension 값이며, work group 의 dimension 을 결정하기도 한다. 이 값은 천둥 CPU 에서는 32 이하, GPU 에서는 16 이하여야 하며, NDIM 이 TDIM 으로 나누어 떨어져야 한다. MAX_SOURCE_SIZE 는 kernel source code 의 최대 크기를 의미하고 KCNT 는 kernel 의 개수를 의미하며, 수정할 필요는 없다. 이 값들을 원하는 대로 수정한 뒤, 천둥에서 실행을 하고 싶다면 make run 을 입력하면 된다.

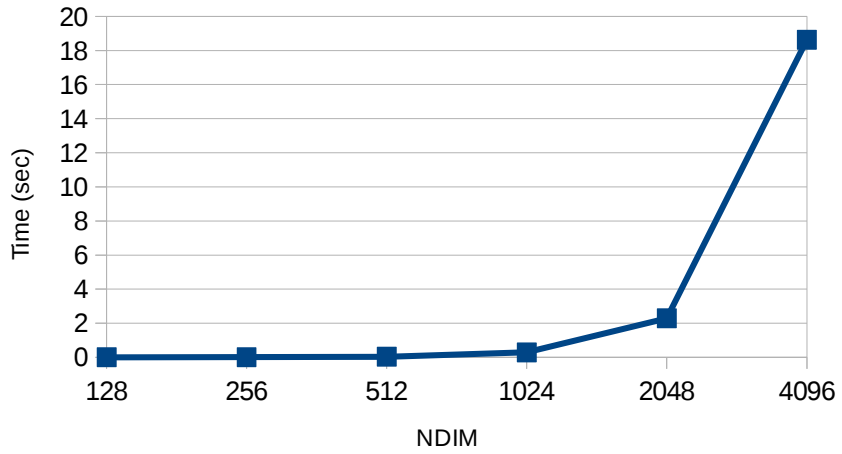
1. 5. 실험 결과

실험은 먼저 행렬의 크기(NDIM)를 증가시켜 가며 수행 시간이 어떻게 변화하는지 살펴보고, 다음에는 행렬의 크기를 적당히 고정해 둔 상태에서 local work size, 즉 tile size(TDIM)를 변화시켜 가며 수행 시간의 변화를 측정하도록 하겠다. Global work size 는 work item 하나가 행렬의 원소 하나를 담당하여 행렬의 크기만큼 고정되어 있기 때문에 이에 의한 성능 변화는 측정하지 않는다.

CPU 를 사용하였을 때 행렬의 크기 변화에 따른 실행시간의 변화는 다음과 같다.

NDIM	Time (sec)
128	0.001988
256	0.009817
512	0.036986
1024	0.289262
2048	2.288624
4096	18.628861

<표 1-1. CPU 에서 행렬의 크기 변화에 따른 실행시간의 변화>



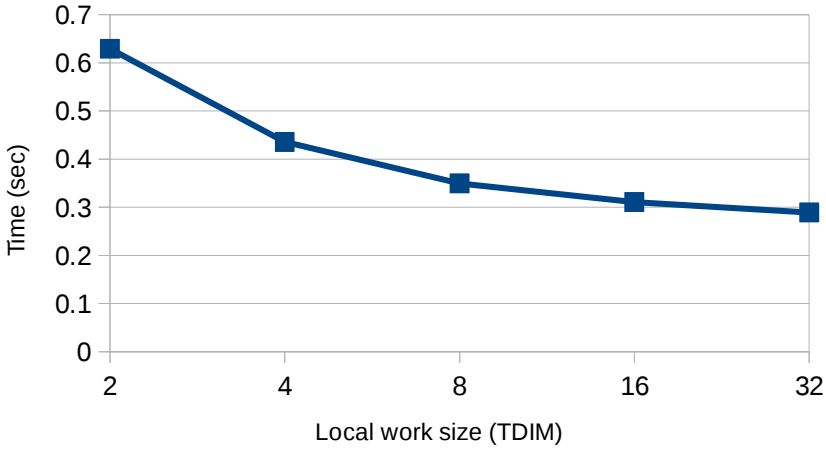
<그림 1-2. CPU 에서 행렬의 크기 변화에 따른 실행시간의 변화>

실험 결과를 보면 행렬의 크기가 증가할 수록 실행시간이 길어진다는 것을 알 수 있다. 간단히 생각하면 NDIM 이 2 배씩 증가할 때마다 전체 계산량은 2^3 = 8 배씩 증가할 것으로 예상할 수 있는데, 실제 실험 결과를 보면 행렬의 크기가 작을 때는 8 배까지 느려지지는 않지만 행렬의 크기가 증가할수록 수행시간의 차이가 8 배에 근접한다는 것을 알 수 있다. 이 때 local work size(TDIM)은 CPU 에서의 최대값인 32 로 고정한 상태에서 실험하였다.

다음으로 CPU 를 사용하였을 때 local work size, 혹은 TDIM 의 변화에 따른 성능 변화를 살펴보겠다. 여기서는 실제 전체 행렬의 크기는 같으므로 실행 시간의 변화가 곧 성능 변화를 의미한다.

TDIM	Time (sec)
2	0.629239
4	0.435732
8	0.349535
16	0.310860
32	0.289177

<표 1-2. CPU 에서 local work size 의 변화에 따른 성능의 변화>



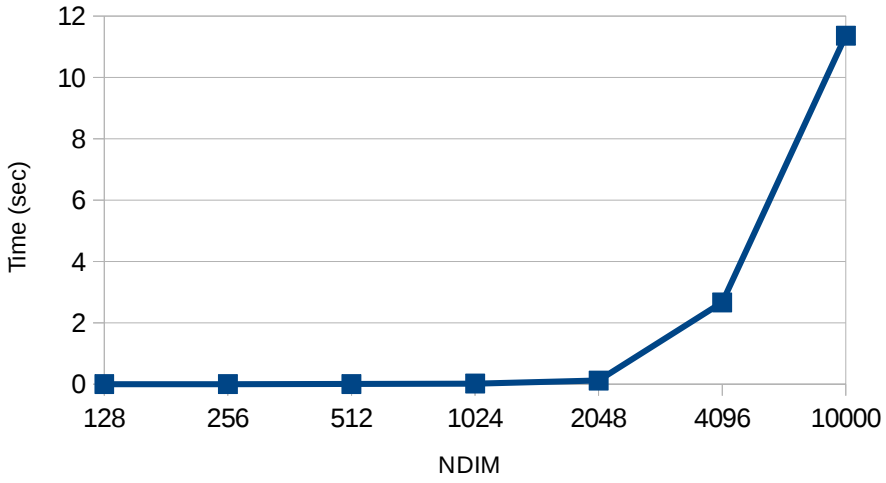
<그림 1-3. CPU 에서 local work size 의 변화에 따른 성능의 변화>

여기서는 local work size, 혹은 tile size 가 증가하면 실행시간은 감소하여 성능은 증가한다는 것을 알 수 있다. 사실 여기에서는 local work size 가 증가하면 tile size 도 증가하여 local memory 의 활용이 증가하므로 local work size 만의 효과라고 보기는 어려우며, tile size 의 증가에 따른 tiling 의 효과 증대도 성능 변화에 큰 영향을 미치고 있다고 볼 수 있다.

다음으로는 GPU 를 사용하였을 때 행렬의 크기 변화에 따른 실행시간의 변화를 알아보겠다. GPU 는 매우 많은 수의 SM(Streaming Multiprocessor) 들이 들어있어 행렬 계산과 같은 병렬화가 잘 되는 계산은 CPU 를 사용하는 것보다 월등한 성능을 기대할 수 있다.

NDIM	Time (sec)
128	0.002150
256	0.002563
512	0.005323
1024	0.020786
2048	0.118353
4096	2.666052
10000	11.358684

<표 1-3. GPU 에서 행렬의 크기 변화에 따른 실행시간의 변화>



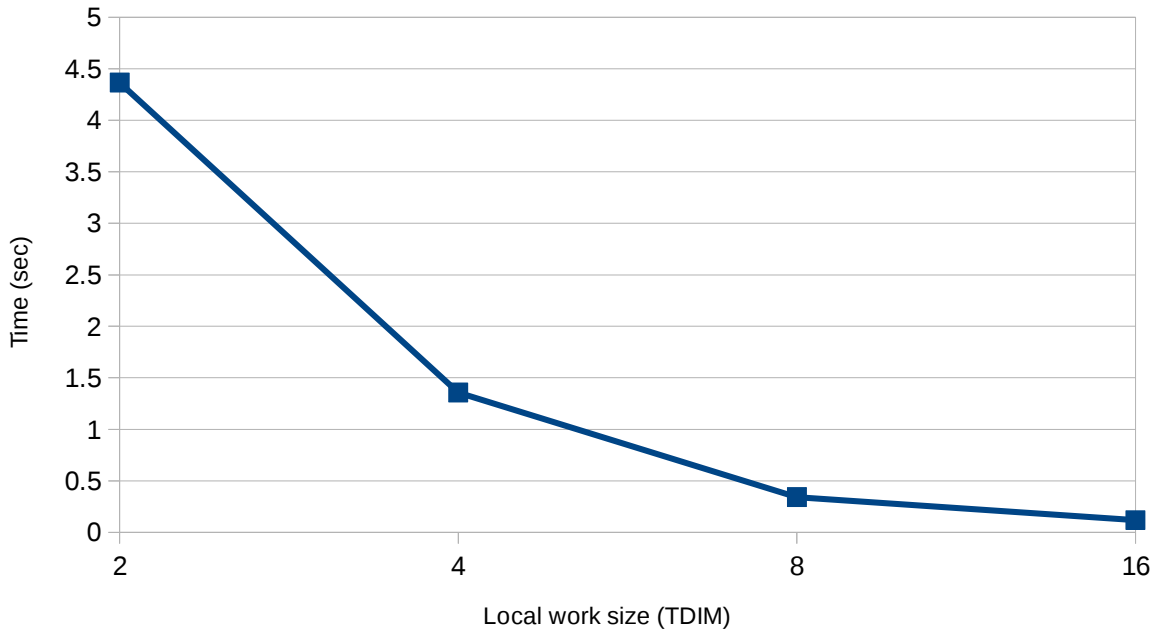
<그림 1-4. GPU 에서 행렬의 크기 변화에 따른 실행시간의 변화>

CPU 와 마찬가지로 GPU 도 행렬의 크기가 증가하면 실행시간도 증가하지만, 행렬의 크기에 따른 시간의 증가폭이 CPU 보다 훨씬 작다는 것을 알 수 있다. 앞에서 예상한대로 병렬화를 exploit 해서 CPU 보다 월등한 성능을 낼 수 있는 것이다.

다음으로는 GPU 에서 local work size, 혹은 tile size(TDIM)의 변화에 따른 성능 변화를 측정해보도록 하겠다. CPU 와 마찬가지로 tiling 에 의한 효과가 커지는 것을 염두해 두어야 한다.

TDIM	Time (sec)
2	4.366497
4	1.357242
8	0.342554
16	0.118414

<표 1-4. GPU 에서 local work size 의 변화에 따른 성능의 변화>



<그림 1-5. GPU 에서 local work size 의 변화에 따른 성능의 변화>

GPU 역시 CPU 와 마찬가지로 local work size 가 증가하면 성능이 증가한다는 것을 알 수 있다. 하지만 CPU 보다 local work size 의 변화에 따른 성능 변화가 훨씬 큰 것을 알 수 있는데, 이는 tile size 에 의한 성능 변화와 더불어 GPU 아키텍쳐 자체가 work group 을 염두에 두고 만들어졌기 때문에 이를 최대한 활용할 수 있기 때문이다.

2. Kmeans

2.1 소스코드 및 구현 방법 설명

Kmeans 에서 OpenCL 관련 세팅을 하는 것은 matmul 과 거의 유사하므로 다시 설명하지는 않도록 하겠다. 소스 파일인 kmeans.cpp 에서 원래는 kmeans() 함수를 iteration 횟수만큼 호출하여 sequential 하게 또는 pthread 를 사용하여 실행하도록 되어 있는데, 여기서 kmeans() 함수 호출 부분을 없애고 iteration 수 만큼의 kernel 호출을 하는 것으로 변경하였다.

CPU 를 사용할 경우는 clCreateBuffer()에서 CL_MEM_USE_HOST_PTR 를 사용하여 host memory 를 그대로 사용할 수 있도록 하였으며, GPU 를 사용할 경우는 CL_MEM_COPY_HOST_PTR 를 사용하여 device memory 에 복사본을 유지하도록 하였다. 여기서 acc_centroids 와 acc_count, 그리고 이들에 대응하는 cl_mem 개체인 d_acc_centroids 와 d_acc_count 는 이후에 kernel 구성에 대해 설명할 때 설명하도록 하겠다.

기존 kmeans 수행 방법은 kmeans() 함수를 불러 iteration 만큼 일련의 계산을 수행하는 것이었는데, 이를 2 개의 kernel 실행과 host 에서의 계산을 포함한 과정을 iteration 만큼 수행하는 것으로 변경하였다. 이렇게 변경한 이유는 기존의 방법을 그대로 유지하여 kernel 1 개에서 모든 계산을 수행할 경우 결과가 제대로 나오기는 하지만, global memory 의 read/write 에 대해 barrier 를 사용해야 하기 때문에 전체 global work size 가 maximum local work size 로 제한이 되어 (barrier 를 이용한 동기화는 한 work group 내에서만 가능하다) 성능이 나오지 않기 때문이다. 따라서 많은 수의 work item 들을 이용하여 병렬화를 할 수 있도록 하기 위해 barrier 가 필요한 부분들을 2 개의 kernel 로 나누어 수행하게 되었다.

첫번째 kernel 인 kernel 0 (kernel0.cl) 은 kmeans 의 가장 처음 과정인 assignment 를 수행한다. 빠른 데이터 접근을 위하여 centroid point 들을 local memory 에 가져오도록 하였으며, 이 과정이 끝난 뒤에 assignement 를 실행해야 하므로 barrier 를 사용하였으나 centroids 자체가 수가 많이 않아 금방 끝나므로 이로 인한 오버헤드는 무시할 만하다. Local memory 로 centroids 를 가져온 후에는 각 work item 이 담당하는 data point 에 대해 assignment 작업을 수행하여 global memory 에 있는

partitioned 를 수정하게 된다.

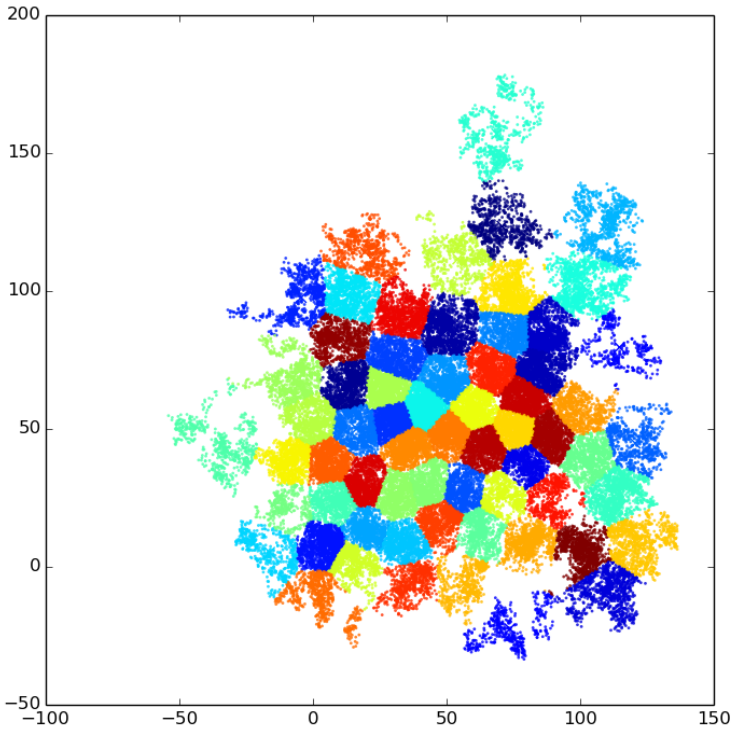
Kernel 0 의 실행이 끝나면 Kernel 1 이 실행되는데, 여기서 아까 언급한 acc_centroids 와 acc_count 가 사용되게 된다. 이 kernel 이 하는 작업은 partitioned 의 값을 index 로 하여 centroids 와 count 를 수정하는 작업인데, 이 partitioned 의 값의 범위가 0 에서 class_n -1 까지이기 때문에 global memory 에 있는 centroids 과 count 를 직접 수정하려 할 경우 여러 work item 이 동시에 수정하는 일이 생길 수 있다. 이를 막고 빠른 계산을 위하여 private memory 에 p_centroids 와 p_count 라는 배열을 생성하여 각 work item 이 계산하는 centroids 와 count 값들을 저장하고, 추후에 global memory 에 위치한 acc_centroids 와 acc_count 중 work item 자신만 사용하도록 할당된 메모리 공간에 아까 계산한 값들을 쓰게 된다. 물론 이때 사용하는 data_i 값들은 work item 마다 independent 하게 할당이 되어 있어 병렬화의 성능을 뽑아낼 수 있다.

Kernel 1 의 실행이 끝나면 kmeans.cpp 의 host program 으로 돌아와 clEnqueueReadBuffer 를 통해 d_acc_centroids 와 d_acc_count 의 device global memory 를 host memory 에 위치한 acc_centroids 와 acc_count 로 읽어들이며, work item 수만큼 퍼져있는 데이터를 서로 더해 accumulation 과정을 수행한다. 이 과정을 수행하면 acc_centroids 와 acc_count 의 가장 앞부분에 sequential 하게 실행했을 때와 같은 centroids 와 count 값이 저장되며, mean point 를 구하기 위해 centroids 에 저장된 sum 을 count 로 나누는 과정까지 host program 이 수행하게 된다. 그 이후 centroids 를 다시 device memory 에 쓰고 iteration 을 반복한다.

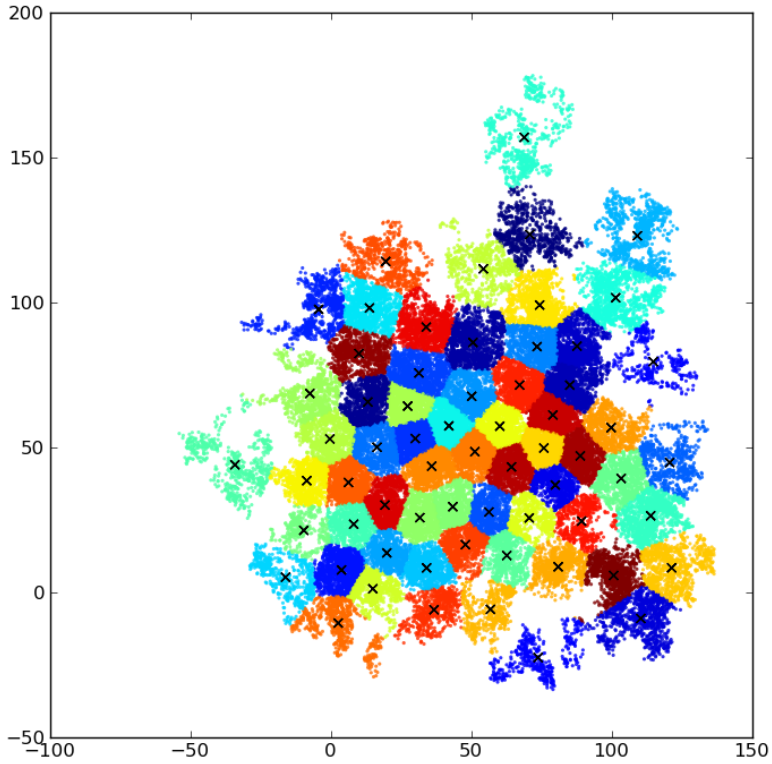
위와 같은 과정을 거치면 sequential 과 같은 결과를 병렬화를 활용하여 훨씬 빠른 시간 내에 얻을 수 있다.

2.2 실험 결과

위에서 구현한 OpenCL 코드가 sequential 한 코드와 같은 결과를 내는지 확인하기 위해 65536 개의 data point 와 64 개의 centroid 를 1024 번의 iteration 을 통해 산출한 결과 그림을 보자.



〈그림 2-1. Sequential 결과〉



〈그림 2-2. OpenCL 결과〉

위 그림들을 살펴보면 OpenCL 로 구현한 코드가 Sequential 로 구현한 코드와 같은 결과를 낸다는 것을 알 수 있다.

이제 CPU 를 사용할 때 global work size 와 local work size 에 따른 성능 변화를 살펴보자. 데이터는 16384 개의 data point 와 16 개의 centroid point 를 사용하였고, 1024 번의 iteration 을 수행하였다. Global work size 를 변화시킬 때는 local work size 를 16 으로 고정하였고, local work size 를 변화시킬 때는 global work size 를 128 로 고정하였다.

GLOBAL_WORK_SIZE	Time (sec)
16	0.9087
32	0.4811
64	0.3347
128	0.4633
256	0.5173

〈표 2-1. CPU 에서 Global work size 에 따른 성능 변화〉

LOCAL_WORK_SIZE	Time (sec)
8	0.4108
16	0.4793
32	0.3660
64	0.6456
128	0.5173

〈표 2-2. CPU 에서 Local work size 에 따른 성능 변화〉

CPU 에서는 global work size 를 증가시킨다고 해서 무조건 성능이 빨라지지는 않으며, 코어 수의 제한이 있어서인지 64 개의 global work size 를 사용할 때 가장 성능이 좋았다. Local work size 도 마찬가지로 증가시킨다고 해서 성능이 좋아지지는 않았고, 32 일때 성능이 가장 좋았다.

다음으로는 GPU 를 사용할 때 global work size 와 local work size 에 따른 성능 변화를 살펴보자. 데이터는 마찬가지로 16384 개의 data point 와 16 개의 centroid pont 를 사용하였고, 1024 번의 iteration 을 수행하였다. Global work size 를 변화시킬 때는 local work size 를 16 으로 고정하였고, local work size 를 변화시킬 때는 global work size 를 1024 로 고정하였다.

GLOBAL_WORK_SIZE	Time (sec)
16	10.4684
32	6.4554
64	4.4692
128	3.6441
256	3.6429

<표 2-3. GPU 에서 Global work size 에 따른 성능 변화>

LOCAL_WORK_SIZE	Time (sec)
16	4.1556
32	4.1069
64	4.0798
128	4.1112
256	4.1032

<표 2-4. GPU 에서 Local work size 에 따른 성능 변화>

GPU 의 경우 global work size 가 증가하면 성능이 좋아지는 것을 볼 수 있었으나, 같은 데이터 사이즈인데도 불구하고 CPU 에 비해 성능이 월등히 느린 것을 확인할 수 있었다. 이는 kmeans 알고리즘 자체가 sequential 한 부분이 많다는 것에서 그 이유를 찾을 수 있다. Local work size 는 성능에 별 영향을 미치지 않는 것을 볼 수 있는데, 이는 mat_mul 과는 달리 work group 에 따른 최적화 방법을 사용하지 않았기 때문으로 생각된다.

마지막으로 1,048,576 개의 data point, 16 개의 centroid, 1024 번의 iteration 을 수행한 결과를 보자. 두 경우 모두 global work size 는 1024, local work size 는 256 으로 고정하였다.

```
[mc21@login0 kmeans]$ cat task*
Time spent: 12.789357641
Time spent: 14.670643255
```

CPU	12.789357641
GPU	14.670643255

전체적인 결과를 보면 CPU 가 GPU 보다 kmeans 알고리즘을 빨리 돌릴 수 있다는 것을 알 수 있으며, 이는 kmeans 알고리즘이 순차적으로 실행되야 하는 부분을 많이 가지고 있어 병렬화에 한계가 있기 때문으로 생각된다.

**** 실행은 make 뒤 make run 을 입력하면 된다.**