[EMDC OS]

멀티테넌트 GPU 환경에서 메모리 자원을 고려한 선제적 동시-스케줄링

버전	내용
v0.1	초안 작성

성균관대학교 컴파일러 및 시스템 연구실

1. 개요

본 문서는 선행 연구인 CASE[1]를 응용하여 멀티테넌트 GPU 환경에서 메모리 자원을 고려한 선제적 동시-스케줄링에 대한 기술문서이다. CASE 스케줄러는 각 태스크가 제공해 주는 메모리 사용량에 대한 정보를 통해 GPU 여유 메모리와 비교를 하여 스케줄 한다. 이때, GPU 여유 메모리 양보다 태스크의 메모리 사용량이 큰 경우 해당 태스크의 스케줄을 지연시켜 메모리 초과 에러(Out of Memory error)를 방지한다. 그렇지만, CASE 스케줄러는 Unified Memory[2]를 고려하지 않는다. 또한, 일반적인 프로그램 상 메모리를 할당 해제하는 부분이 코드의 맨 아랫부분에서 진행되어 사용이 끝난 메모리 변수도 불필요하게 메모리를 점유하고 있는 상황 역시 고려하지 않는다. 따라서, 본 문서는 기존 CASE 스케줄러를 응용하여 Unified Memory 를 지원하고 수명이 다 한 변수를 그 즉시 할당 해제하여 GPU의 여유 메모리를 최대한으로 활용하는 스케줄링 기법을 제시한다.

2. 배경 지식

1) CASE: Compiler Assisted SchEduling Framework

CASE 스케줄러는 프로그램 내 GPU operations 들을 묶어 하나의 태스크를 구성한다. CASE 는 컴파일러를 통해 태스크의 시작 지점과 종료 지점에 코드를 삽입하고 해당 코드를 통해 스케줄러와 태스크가 정보 전달을 한다. 이때, 정보 전달은 shared memory 를 통해 진행된다. 정보 전달을 통해 스케줄러가 각 태스크별 메모리 사용량을 알게 되고 현재 디바이스의 여유 메모리와 비교한다. 만일, 현재 디바이스 여유 메모리보다 디바이스의 요구 메모리 양이 더 작다면 해당 디바이스에 태스크를 스케줄 한다. 반대로 디바이스의 요구 메모리 양이 더 크다면 해당 태스크를 스케줄 하지 않고 대기함으로써 Out of Memory 에러를 방지한다.

2) NVIDIA Unified Memory

NVIDIA 에서는 프로그래머들에게 편의성을 제공하기 위해 NVIDIA Unified Memory 를 도입했다. Unified Memory 는 49-bit 의 가상 메모리 주소와 page fault 시스템을 사용한다. 따라서, 기존 GPU pinned memory 와는 다르게 별도로 데이터를 copy 하지 않고 사용할 수 있다. 또한, GPU 디바이스의 메모리 사이즈보다 더 많은 양을 사용할 수 있고 이 기술을 over-subscription 이라고 한다. 하지만, on-demand paging 방식을 통해 page fault 시스템을 사용하게 되면 fault handling 을 하는 시간 때문에 기존 GPU Pinned Memory 방식보다 성능이 저하되게 된다. 따라서, cudaMemAdvise() API 에 적절한 flag 를 사용하여 Unified Memory 의 페이지 위치 및 접근 방식을 조절하는 방식으로 성능을 보완한다.

3. GPU Pinned memory 와 Unified Memory 성능 비교

Unified Memory 기반의 태스크를 사용하기 위해 Unified Memory 기반 태스크가 기존 GPU Pinned memory 기반 태스크와 동일하게 동작하도록 몇 가지 옵션을 설정하였다. 기존 cudaMalloc() 함수를 호출하여 할당한 메모리는 GPU 디바이스 쪽에 고정되지만 별다른 옵션을 설정하지 않고 cudaMallocManaged() 함수를 호출하여 할당한 메모리는 GPU 디바이스 쪽에 고정되지 않는다. 따라서, cudaMemAdvise() 함수에 setPreferredLocation 플래그를 추가하고 위치를 디바이스 쪽으로 설정하여 GPU 디바이스 쪽에 Unified Memory를 고정시켜 둔다. 실제로 동일하게 동작하는지 보기 위해 구간을 나누어 시간을 비교 분석해 보았다. 실험에 사용한 응용은 Polybench[3]의 2MM과 2DCONV를 사용하였다.

그림 1을 보면 Unified Memory 기반 태스크도 GPU 디바이스 쪽에 페이지를 고정시켜두었기 때문에 커널이 접근하는 시간은 동일하게 소요된다. 그뿐만 아니라, 데이터를 전송하는 과정에서 기존 GPU Pinned memory 기반 태스크의 경우 호스트 쪽 pageable memory에서 호스트 쪽 pinned memory로 한번 복사를 한 후 GPU pinned memory로 복사하는 과정을 거치지만 Unified Memory 기반 태스크의 경우 바로 디바이스 쪽으로 데이터를 전송하기 때문에 시간적 이점이 있다. 실제로 그림1의 MemoryTransfer(HtoD) 부분을 보면 Unified Memory가 시간적 이점이 있고 cudaMallocHost로 호스트 쪽에 페이지를 고정시켜 둔 방식과 비교해보았을때는 동일한 시간이 소요되는 것을 볼 수 있다.

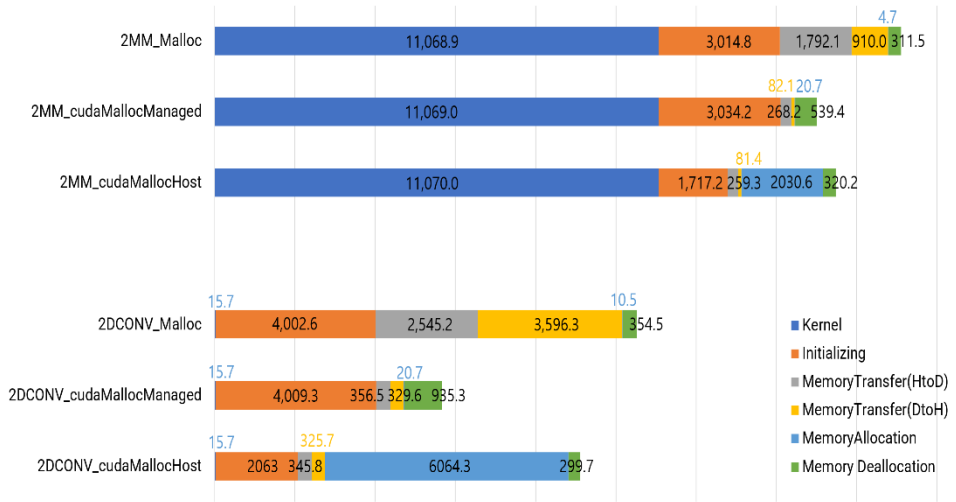

그림 1 GPU pinned memory 기반 태스크와 Unified Memory 기반 태스크의 수행 시간 비교 (밀리초)

4. Unified Memory page 위치 및 접근 방식

cudaMemAdvise()와 cudaMemPrefetchAsync() API 를 통해 Unified Memory 의 페이지 위치 및 접근 방식을 설정할 수 있다. 가능한 페이지 위치 및 접근 방식은 다음과 같다. (현재 호스트 쪽에서 initialize 되었다고 가정한다.)

Zero-Copy (CPU Pinned)

cudaMemAdvise(A, size, cudaMemAdviseSetPreferredLocation, cudaCPUDeviceID)

cudaMemAdvise(A, size, cudaMemAdviseSetAccessedBy, 0)

GPU-Pinned

cudaMemAdvise(A, size, cudaMemAdviseSetPreferredLocation, 0)

cudaMemPrefetchAsync(A, size, 0, 0)

Read-only Copy

cudaMemAdvise(A, size, cudaMemAdviseSetReadMostly, 0)

cudaMemPrefetchAsync(A, size, 0, 0)

위와 같은 페이지 옵션을 설정하여 Unified Memory 의 over-subscription 기법을 사용하게 되는 태스크의 페이지 위치 및 접근 방식을 더 나은 성능이 발생하도록 설정해 줄 수 있다.

5. 선제적 스케줄링 기법

1) Eager Free

Eager Free 는 수명이 다 한 변수를 그 즉시 할당 해제하는 기법이다. 더 이상 사용되지 않는 메모리를 그 즉시 할당 해제하고 해당 변수가 점유하던 메모리 사용량을 스케줄러에게 전달함으로써 메모리 자원을 효율적으로 사용할 수 있게 된다. Eager Free 는 기존 태스크의 종료를 알리는 bemps_free 외에 pre_bemps_free 라는 별도의 함수를 메모리 변수들의 수명이 다하는 지점에 삽입함으로써 실행된다.

2) Eager Launch

Eager Launch는 Unified Memory를 통해 메모리 초과 사용(over-subscription)이 가능하다. 그로 인해 가용한 메모리 양이 현재 태스크의 메모리 요구 양보다 작은 경우에도 스케줄이 가능하게 된다. bemps_begin()을 통해 스케줄러에게 태스크의 총 메모리 사용량을 전달하면 스케줄러가 현재 GPU 여유 메모리 양과 비교하고 만일 GPU 여유 메모리 양이 더 작다면 일부 메모리 양만 리턴하여 태스크가 일부 메모리만 할당받은 상태로 Eager Launch되어 실행된다.

Eager Launch된 태스크도 디바이스의 여유 공간이 생기게 되면 메모리를 추가적으로 확보 받아야 하기 때문에 앞선 태스크가 종료되었거나 Eager Free를 통해 메모리를 할당 해제한 태스크가 있는지 확인하여야 한다. 앞선 태스크의 종료 및 Eager Free가 실행되었는지는 현재 디바이스의 여유 메모리 양에 변동이 생겼는지를 확인하는 함수를 통해 알 수 있다. 해당 함수가 호출되면 메모리 양의 변동에 따라 Eager Launch 태스크에게 추가적으로 할당된 메모리가 있는지 그 양을 리턴해준다. 만일, Eager Launch된 태스크가 종료되기 전 메모리 여유 공간이 생긴다면 현재 커널이 실행 중인 스트림과 별도의 스트림에서 메모리를 전송하여 메모리를 확보 받을 수 있다.

한 디바이스에 Eager Launch 되는 태스크의 수가 많아지면 Eager Launch

태스크 사이의 메모리 간섭과 over-subscription ratio가 증가하여 성능 저하가 발생하게 된다. 따라서, 각 디바이스 별 Eager Launch 태스크는 1개로 제한한다.

아래 그림 2와 3은 동일한 순서의 응용으로 구성된 워크로드를 1대의 GPU에서 CASE와 CASE+Eager Launch 기법 각각에서 실행하였을 때 각 응용이 소요한 시간을 나타낸다. Pending time은 태스크가 각 기법의 스케줄링 정책에 따라 스케줄 되지 못하고 대기중인 시간을 의미한다. 그림 3의 화살표 부분을 보면 Eager Launch가 정상적으로 동작한 것을 확인할 수 있다.

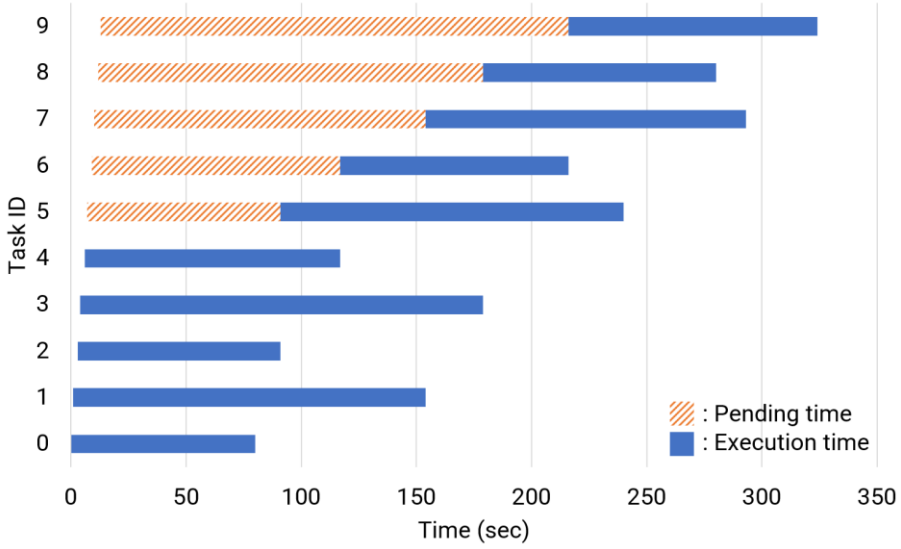

그림 2 CASE 기법에서의 응용 별 실행 시간 (초)

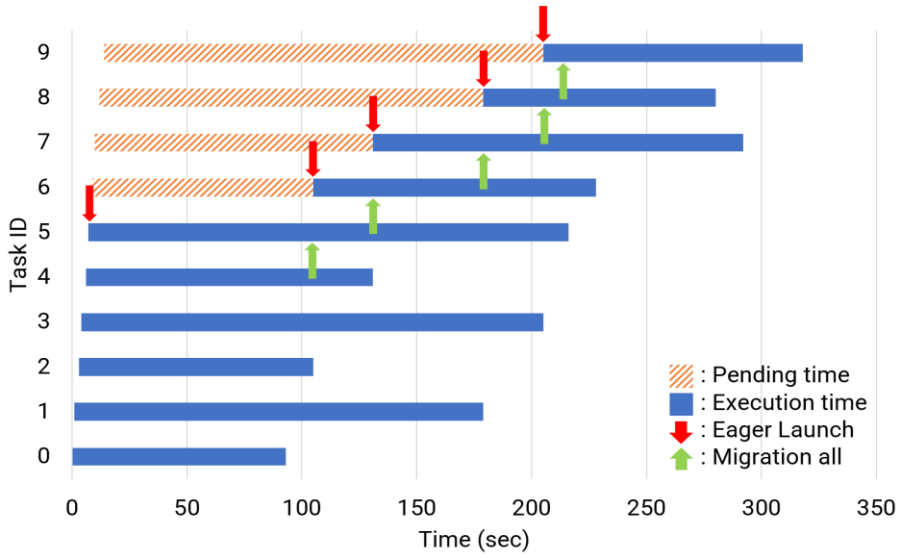

그림 3 CASE+Eager Launch 기법에서의 응용 별 실행 시간 (초)

참고문헌

[1] C. Chen, et al. "CASE: a compiler-assisted scheduling framework for multi-gpu systems," in PPoPP, pp. 17-31. 2022.

[2] "CUDA C++ Programming Guide" https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[3] S. Grauer-Gray, et al. "Auto-tuning a High-level Language Targeted to GPU Codes" in InPar. 2012.