[EMDC OS]

멀티테넌트 GPU 환경에서
메모리 자원을 고려한
선제적 동시-스케줄링

버전	내용
V1.0	최초 배포판

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. 하지만, CASE 스케줄러는 Unified Memory[2]를 지원하지 않는다. 그에 따라, GPU 메모리 자원을 최대한으로 활용하지 못하게 되고 그로 인해 성능 향상 기회를 놓치게 된다. 따라서, 본 문서는 기존 CASE 스케줄러를 응용하여 Unified Memory 를 지원하고 그 때 발생할 수 있는 문제점들을 해결하여 GPU 의 처리량 및 이용량을 향상시키는 Eager Launch 스케줄러를 제안한다.

2. 배경 지식

1) CASE: Compiler Assisted SchEduling Framework

현재, 효율적인 GPU 자원 사용을 통한 GPU 이용량 및 처리량 향상을 위해 멀티테넌트 환경에서 GPU 를 공유하여 사용하고 있다. 그렇지만, 각 테넌트 별 메모리 사용량에 대한 정보가 없이 GPU 를 공유하여 사용하게 되면 Out-of-Memory Error 가 발생할 수 있고 이는 심각한 성능 저하를 초래한다.

선행 연구에서는 Out-of-Memory Error 를 방지하기 위해 각 태스크의 메모리 사용량에 대한 정보를 사용한다. 스케줄러와 태스크 간의 정보 전달을 위해 컴파일러는 별도의 코드를 삽입한다. 추가로, 컴파일러를 통해 해당 태스크가 사용하는 메모리 양을 산출할 수 있고, 태스크는 삽입된 코드를 통해 산출된 메모리 양에 대한 정보를 공유 메모리를 사용하여 스케줄러에게 전달한다. 스케줄러는 전달받은 정보와 현재 GPU 의 여유 메모리 양을 비교하여 태스크의 메모리 요구량을 충족하는 GPU 가 존재할 경우 해당 GPU 에 태스크를 스케줄한다. 만일, 태스크의 메모리 요구량을 충족하는 GPU 가 존재하지 않을 경우, 메모리 요구량을 충족하는 GPU 가 생길 때까지 태스크의 스케줄을 지연시킴으로써 Out-of-Memory Error 를 방지한다.

2) 통합 메모리

통합 메모리(Unified Memory)는 기존의 프로그래밍 모델에서 CPU 와 GPU 간의 메모리 접근을 단순화하기 위해 개발된 메모리 관리 기술이다. 그림 1 을 보면 알 수 있듯이 통합 메모리는 CPU 와 GPU 가 공유하는 단일 주소 공간을 제공하여, CPU 와 GPU 간의 데이터 전송 및 메모리 할당의 복잡성을 줄여준다. 이 기술을 통해 프로그래머는 GPU 에 데이터를 명시적으로 복사하지 않고도 CPU 와 GPU 간에 데이터를 효율적으로 공유할 수 있다.

기존 프로그래밍 모델에서는 프로그래머가 직접 데이터를 GPU 혹은 CPU 쪽으로 전송하여 사용한다. 그렇지만, 통합 메모리는 하나의 가상 주소(Unified Virtual Memory)를 기반으로 동작하기 때문에 별도의 명시가

필요하지 않게 된다. 통합 메모리 환경에서는 만일 접근하려는 페이지가 해당 디바이스 쪽에 위치하지 않을 경우 페이지 폴트(page fault)가 발생한다. 페이지 폴트가 발생함에 따라 page migration 을 통해 필요한 데이터가 접근하려는 디바이스 쪽으로 옮겨진다. 이 경우 메모리 공간이 충분하다면 단순하게 page migration 이 발생하고, 그렇지 않다면 기존 메모리에서 페이지를 eviction 을 시켜 여유 공간을 만들어 준 후 migration 을 진행한다. 해당 기술을 메모리 초과 사용(over-subscription)이라 하며, 메모리 초과 사용을 통해 GPU 전체 메모리보다 더 큰 메모리를 요구하는 응용을 에러 없이 실행시킬 수 있게 된다.

앞서 언급한 것처럼 통합 메모리를 사용하게 될 경우, 프로그래머의 부담을 해소하고 효율적인 데이터 공유를 기대할 수 있다. 그렇지만 통합 메모리에서의 CPU 와 GPU 간 데이터 전송은 PCIe 인터페이스를 통해 이루어지기 때문에, 빈번한 페이지 폴트가 발생할 경우 데이터 전송 오버헤드가 누적되어 심각한 성능 저하를 초래할 수 있다는 단점이 있다.

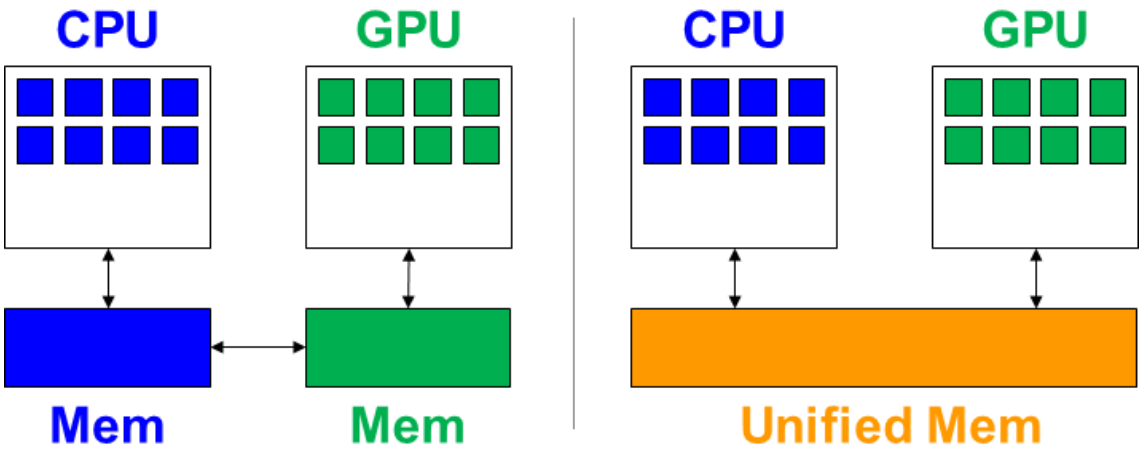

그림 1 프로그래머 관점에서의 통합 메모리

3. Eager Launch 스케줄러

통합 메모리 지원을 위해 태스크의 메모리 할당 방식을 기존 GPU 고정 메모리 할당에서 통합 메모리 할당으로 변경해 주어야 한다. 하지만, 단순히 메모리 할당 방식만 바꿔주게 되는 경우 기존 데이터 전송 과정이 삭제되고 페이지 폴트와 요구 페이징(demand paging)의 방식으로만 데이터가 전송되게 되어 성능 저하가 발생한다. 따라서, 몇 가지 CUDA Runtime API를 추가로 호출해 성능 저하 요소를 없앤다.

기존 태스크 코드는 cudaMalloc() API를 통해 디바이스 메모리를 할당하는데 이 경우 디바이스 메모리에 할당되는 메모리는 GPU 고정 메모리이다. 따라서, 통합 메모리 지원을 위해 cudaMallocManaged() API만 사용하는 것이 아닌, 추가로 페이지 옵션을 설정해 주어야한다. 해당 페이지 옵션은 cudaMemAdvise() API에 setPreferredLocation 플래그를 인자로 전달해줌으로써 가능하다. 또한, 기존 태스크 코드에서는 cudaMemcpy() API를 호출해 별도로 데이터를 전송을 해주는데 통합 메모리 상에서 별다른 데이터 전송 API를 호출하지 않을 경우, 데이터는 모두 요구 페이징 방식으로 전달된다. 따라서, 통합 메모리 상에서도 cudaMemPrefetchAsync() API를 호출하여 데이터를 전송해주어야 한다.

Eager Launch 되어 실행되는 태스크는 메모리 요구량보다 적은 양의 메모리만을 확보 받은 채로 실행된다. 따라서, Normal Launch 태스크의 종료에 따라 디바이스에 여유 메모리가 발생한다면, 스케줄러는 현재 스케줄 대기중인 태스크에게 해당 메모리를 부여해주는 것이 아닌, 일부 메모리만 확보 받아 실행중인 Eager Launch 태스크에게 우선적으로 메모리를 확보해 주어야 한다. 해당 작업을 위해 스케줄러는 태스크에게 추가 메모리 양을 명시해주고 만약 해당 양이 태스크의 메모리 요구량을 충족한다면 태스크는 현재 실행중인 커널과 별도의 커널에서 데이터를 전송함으로써 커널 실행과 데이터 전송을 오버랩하여 실행하게 된다.

Eager Launch를 통해 태스크의 스케줄 대기 시간을 줄여 스케줄 되는 시점을 앞당기게 되면 같은 시점에서 한 디바이스에 동시-스케줄되는 태스크의 수가 많아지게 된다. 동시-스케줄되는 태스크의 수가 많아짐에 따라 태스크 간의 GPU 자원에 대한 간섭이 발생할 수 있다. 만약 Normal Launch 태스크의 커널 런치에 앞

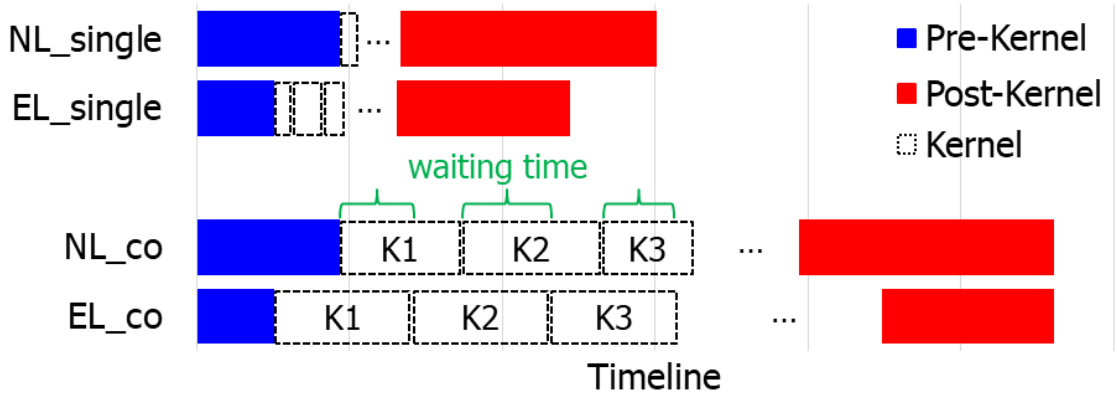

그림 2 Eager Launch 태스크의 Normal Launch 태스크에 대한 간섭

서 Eager Launch 태스크의 커널이 먼저 런치가 되어 실행되는 경우, GPU 컴퓨팅 자원을 확보 받아 스케줄 된 Normal Launch 태스크의 자원을 Eager Launch 태스크가 선점하여 사용하게 되는데 이 때 Eager Launch 태스크의 메모리 초과 사용 비율이 높다면 Normal Launch 태스크의 페이지를 호스트쪽으로 eviction 하여 성능 저하가 발생하게 된다.

그림 2는 앞서 예시로 든 성능저하 상황을 나타낸다. 그림 2에서 맨 위 2줄은 Normal Launch 태스크와 Eager Launch 태스크가 각각 단일로 실행되었을 때의 타임라인을 나타낸다. 그림 2를 보면 알 수 있듯이 각 응용은 여러 개의 커널로 구성이 되어있다. 아래 2줄은 두 태스크가 한 디바이스에 동시-스케줄 되어 실행될 때의 타임라인을 나타내는데, 메모리 요구량의 차이와 같은 이유로 Eager Launch 태스크의 커널 전 작업 소요 시간이 더 적게 걸리는 것을 확인할 수 있다. 이 경우, Eager Launch 태스크의 커널이 먼저 런치 되어 실행된다. Normal Launch 태스크의 커널이 런치되었을 때, GPU 컴퓨팅 자원을 Eager Launch 태스크가 먼저 선점하여 사용중이기 때문에 Normal Launch 태스크의 커널은 스케줄 대기 상태에 놓이게 된다. 그에 따라 Normal Launch 태스크의 커널 실행 시간에는 단순 커널 실행 시간뿐만 아니라 여유 GPU 컴퓨팅 자원이 생길때까지 대기하는 시간이 포함되게 되고 이 대기 시간은 Eager Launch 태스크의 커널 시간이 늘어날수록 증가하게 된다. 실제로, Rodinia 벤치마크 중 sradv1 응용의 메모리 요구량을 조절하여 실험을 진행해보면 태스크 각각이 40배 정도의 성능저하가 발생하는 것을 확인할 수 있다.

4. Normal Launch 태스크에 대한 간섭 최소화

1) Priority-method

CUDA 프로그래밍 모델에서 GPU 관련 연산들은 스트림 내에서 정렬되어 실행된다. 이때 스트림이란, 코드 내에서 기술된 순서대로 디바이스에서 동작하는 일련의 연속된 연산을 의미하며 스트림 내의 연산들은 FIFO 순서로 실행된다. 하지만, GPU 자원의 여유가 있다면, 다른 스트림에서의 동작들이 서로 오버랩되어 실행이 가능해진다. 따라서, 별도의 스트림에서 실행되고 있는 Normal Launch 태스크의 작업과 Eager Launch 태스크의 작업이 동시에 실행되는 것이 가능하다.

그렇지만, 만일 GPU 자원이 충분하지 않다면, Eager Launch 태스크가 GPU 자원을 모두 사용하며 실행이 되어 Normal Launch 태스크가 실행되지 못하고 대기하게 되는 상황이 발생한다. 두 태스크의 커널이 속해있는 스트림이 만일 우선순위가 같다면, SM 은 지속적으로 Eager Launch 태스크의 커널을 스케줄하여 실행할 수가 있고 Normal Launch 태스크의 대기 시간은 더 길어지게 된다. 이와 같은 성능 저하를 막기 위해 본 논문에서는 태스크별로 우선순위를 부여하여 Normal Launch 태스크에 대한 간섭을 최소화한다.

태스크별 우선순위는 CUDA Runtime API 인 cudaStreamCreateWithPriority()를 통해 부여된다. 태스크는 스케줄러로부터 부여받은 메모리양과 자신의 메모리 요구양을 비교해 Normal Launch 태스크인지, Eager Launch 태스크인지 알 수 있다. 이때 만약 Normal Launch 태스크라면 높은 우선순위의 스트림을 생성하고 그렇지 않다면 낮은 우선순위의 스트림을 생성한다. 스케줄 요청 당시에는 GPU 디바이스 여유 메모리가 충분하지 않아 스케줄러에 의해 Eager Launch 되어 실행된 태스크일지라도 응용 실행 중에 메모리를 추가로 확보받아 Normal Launch 태스크가 될 수 있다. 이 경우, 초기에 생성한 낮은 우선순위 스트림에서 이후의 커널이 실행되는 것을 방지하기 추가 메모리 확보 모니터링 API 에서 우선순위를 변경해준다. 이후 실행될 커널들은 우선순위가 조정된 스트림에서 실행되기

때문에 추후에 스케줄 될 Eager Launch 태스크의 커널보다 높은 우선순위로 GPU 자원을 선점해 실행된다.

2) Barrier-method

앞서 Normal Launch 태스크에 대한 Eager Launch 태스크의 간섭을 최소화하기 위해 각 태스크별로 우선순위를 설정했다. 하지만, 우선순위 설정에도 불구하고 각 태스크의 첫 커널 런치 시점에 따라 Eager Launch 태스크가 GPU 자원을 선점하는 것은 예방하기 어렵다. Eager Launch 태스크의 커널이 먼저 런치된 시점에서 아직 Normal Launch 태스크의 커널이 런치된 것은 아니기 때문에 SM 의 입장에서는 Eager Launch 태스크의 커널을 먼저 스케줄할 수밖에 없기 때문이다. 따라서, Eager Launch 태스크의 첫 커널 런치 전에 별도의 배리어를 설정하여 Eager Launch 태스크의 커널들이 모두 런치된 후에 실행되도록 한다.

배리어 설정을 위해 기존 태스크 및 스케줄러 내에서 몇 가지 수정 사항이 발생한다. 태스크 내에서는 첫 번째 커널 런치 전, 만일 Eager Launch 태스크라면 스케줄러로부터의 배리어 해제 정보가 오기 전까지 기다리는 작업이 추가된다. 또한, Normal Launch 태스크인 경우, 마지막 커널 런치 후 스케줄러에게 태스크 내의 모든 커널 런치가 완료되었다는 정보를 전달하는 작업이 추가된다. 스케줄러는 각 GPU 디바이스별로 Normal Launch 태스크의 커널 런치 완료 상황을 관리할 수 있고, Normal Launch 태스크로부터의 정보를 바탕으로 해당 GPU 디바이스 내에 모든 Normal Launch 태스크의 커널 런치가 완료되었다면 같은 디바이스에 Eager Launch 된 태스크에게 배리어 해제 정보를 전달한다.

그림 3 은 우선순위 및 배리어 설정에 따른 Eager Launch 태스크의 동작 방식과 선행 연구 대비 얻을 수 있는 성능 향상 기댓값을 나타낸다. 선행 연구에서는 GPU 여유 메모리 부족으로 인해 3 번째 태스크가 2 번째 태스크가 종료될 때까지 스케줄이 지연된다. 그렇지만 우선순위 및 배리어를 설정한 Eager Launch 스케줄러에서는 스케줄 대기 시간 없이 곧바로 스케줄이 가능하다. 이때, Normal Launch 태스크에 대한 간섭을 최소화하기 위해 앞서

스케줄된 Normal Launch 태스크들의 모든 커널들(첫번째 태스크에서의 K3, 두번째 태스크에서의 K2)이 런치 될 때까지 Eager Launch 태스크는 배리어를 통해 대기하게 된다. 배리어가 해제된 후에도 높은 우선순위인 Normal Launch 태스크들의 커널들이 GPU 자원을 선점하여 실행되고 여유 GPU 자원이 생기는 시점에 Eager Launch 태스크의 커널이 실행된다. 그에 따라, Eager Launch 스케줄러에서도 Normal Launch 태스크들이 기존 스케줄러와 비교했을 때 큰 성능 저하 없이 실행되게 되고 그림 3 에 명시한 만큼의 성능 향상을 기대할 수 있게 된다.

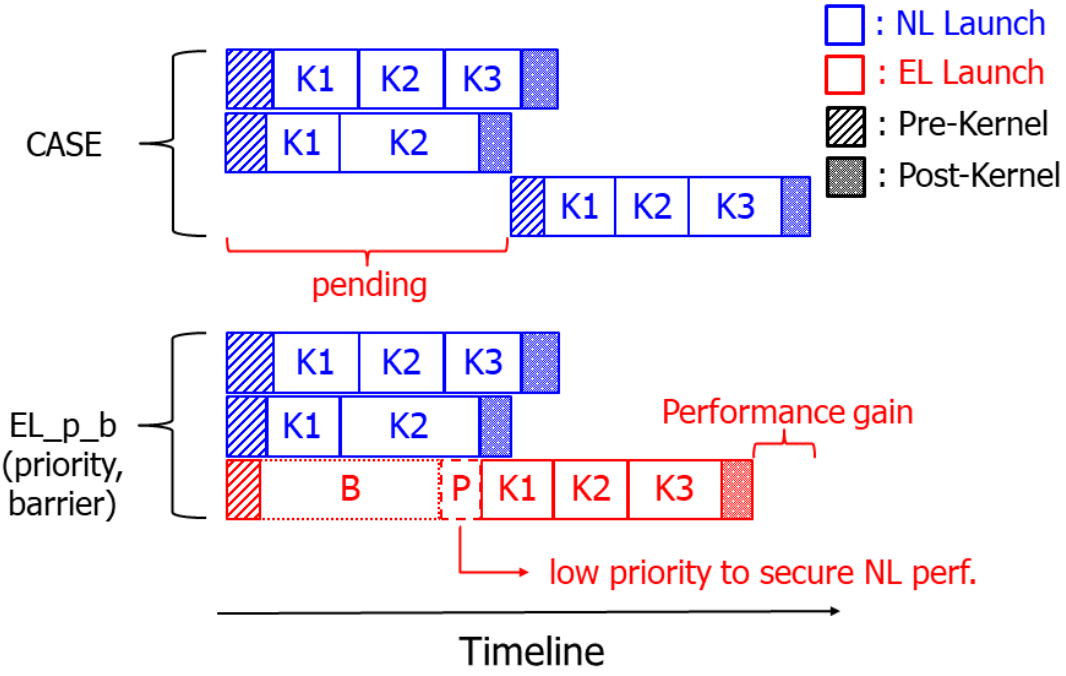

그림 3 우선순위 및 배리어 설정 시 Eager Launch 태스크의 동작 방식

5. Remedies for Priority-, Barrier- methods

1) Set OS-Ratio Threshold

태스크 별로 우선순위를 설정한 Priority-method 의 경우, 높은 우선순위의 태스크와 낮은 우선순위의 태스크가 모두 쓰레드 블록 스케줄링 대기 중일 경우, 의도한 대로 높은 우선순위 태스크의 쓰레드 블록이 GPU 의 SM 에 먼저 스케줄 되어 실행되게 할 수 있다. 하지만, 만일 높은 우선순위의 태스크가 여러개의 커널로 구성되어 있고, 각 커널 사이에 host processing 과 같은 작업이 포함되어 있다면, 높은 우선순위 태스크의 쓰레드 블록 대신 낮은 우선순위 태스크의 쓰레드 블록이 스케줄되어 실행된다. 이 때, 낮은 우선순위 태스크가 높은 OS-Ratio(over-subscription ratio)에서 실행된다면, 해당 태스크의 커널 실행 시간이 증가하게 되고, host processing 작업 후에 런치된 높은 우선순위 태스크의 커널이 바로 스케줄되어 실행되는 것이 아니라 낮은 우선순위 태스크의 커널 실행 시간만큼 대기하게 된다. 또한, 통합 메모리 특성에 의해 낮은 우선순위 태스크의 커널이 실행되는 동안 높은 우선순위 태스크의 페이지가 호스트쪽으로 eviction 될 수 있어 높은 우선순위 태스크의 커널 실행 시간이 더 증가하게 된다.

위와 같은 성능저하를 방지하기 위해 낮은 우선순위 태스크의 커널 같은 경우, 각 커널별 메모리 실 사용량을 기반으로 임계 OS-Ratio 값을 설정한다. 임계 OS-Ratio 를 설정함에 따라, 낮은 우선순위 태스크의 커널은 높은 우선순위 태스크의 자원을 간섭하지 않는 선에서 실행될 수 있다. 통합 메모리를 사용하게 되면, 현재 접근하려는 페이지가 호스트 쪽에 위치한 경우, 해당 페이지를 디바이스 쪽으로 위치시켜 접근하는 요구 페이징(demand-paging)이 발생하기 때문에 한 커널이 실행될 때 필요한 메모리 공간이 커널의 인자로 받는 메모리 오브젝트의 크기보다 작을 수 있다. 만약, 커널이 사용하는 메모리 오브젝트 크기를 기준으로 임계 OS-Ratio 값을 설정하게 되면, 실제 필요한 공간보다 더 많은 공간이 필요한 것으로 스케줄러가 인식하여 자원을 비효율적으로 사용하게 된다. 따라서, 별도의 작업을 통해 각 응용의 커널별 메모리 실 사용량을 구해 스케줄러에게 전달해준다. 이 때,

App	Kernel	Static Data Size	Memory Footprint
backprop	Kernel #1	4.5GiB	128MiB
	Kernel #2	8.8GiB	78MiB
sradv1	Kernel #1	1.5GiB	1.5GiB
	Kernel #2	4.5GiB	4.5GiB
	Kernel #3	3.0GiB	3.0GiB
	Kernel #4	8.9GiB	8.9GiB
	Kernel #5	8.9GiB	8.9GiB
	Kernel #6	1.5GiB	1.5GiB

표 1 응용의 커널별 메모리 오브젝트 크기 및 메모리 실 사용량 비교

별도의 작업이란, 각 응용의 커널 실행전마다 모든 페이지를 호스트 쪽에 위치시켜두고, 커널 실행 전 후의 디바이스 메모리 양을 비교하는 것을 의미한다. 표 1 은 Rodinia 벤치마크 중 일부 응용들의 커널별 메모리 오브젝트 크기 및 메모리 실 사용량을 나타내는 표이다. 표 1 을 보면 알 수 있듯이 어떤 응용은 메모리 오브젝트 크기와 메모리 실 사용량의 차이가 크게 발생하고, 다른 응용은 그렇지 않은 것을 확인할 수 있다.

2) Launch Kernels from Eager Launch Task when GPU Idle

배리어 설정을 통해 Normal Launch 태스크의 모든 커널이 런치된 후에 Eager Launch 태스크의 커널이 런치 되게 하였을 경우, Normal Launch 태스크에 대한 Eager Launch 태스크의 간섭이 발생하지 않아 Normal Launch 태스크의 성능을 보장해줄 수 있다. 하지만, Normal Launch 태스크의 커널 전 작업의 소요 시간이 길다면, 해당 시간 동안은 GPU가 idle 함에도 불구하고 실행되고 있는 커널이 없어 GPU의 처리량이 감소된다. 따라서, 현재 Normal Launch 태스크들의 커널이 런치되지 않은 상황이라면 Eager Launch 태스크의 커널을 실행시키는 것이 GPU 처리량 향상에 도움이 된다.

위와 같은 작업은 태스크와 스케줄러간의 추가 정보 전달을 통해 가능하다. Normal Launch 태스크는 만일 커널을 런치했다면 스케줄러에게 커널 런치에 관련한 정보를 전달해준다. 스케줄러는 해당 정보가 존재하지

않는다면 Eager Launch 태스크에게 커널 런치를 진행하라는 정보를 전달하고, 만약 존재한다면 Eager Launch 태스크에게 다음 커널 런치는 대기하라는 정보를 전달해준다. Eager Launch 태스크의 경우, 각 커널마다 스케줄러로부터 온 정보를 확인해 커널의 런치 여부를 정한다.

6. 성능 평가

본 기술의 성능 평가는 Intel Xeon Gold 6230 CPU, NVIDIA V100 16GB 4대, 1TB DRAM이 탑재된 서버에서 진행되었다. CUDA 버전은 12.2이며, NVIDIA Driver 버전은 535.183.01이다.

성능 평가에는 선행 연구와 동일한 Rodinia 벤치마크가 사용되었으며, 그 중, backprop(패턴 인식), sradv1, sradv2(이미지 처리), lavaMD(분자동역학), needle(생물정보학), dwt2d(이미지/비디오 압축), bfs(그래프 탐색) 등이 사용되었다. 또한, 각 응용 실행 시 command line arguments 들을 조정하여 응용 별 메모리 사용량을 조절해 주었다. 이때, 메모리 사용량이 4GB를 초과한다면 Large 응용, 1GB 이상 4GB 이하로 형성된다면 Small 응용으로 분류하였다. 또한, 형성되는 Large, Small 응용이 두 개 이상이라면, 올림차순으로 정렬해 1, 2, .. 등으로 명칭하였다. 앞서 설명한 분류 방법을 토대로 총 17개의 응용을 생성해 응용 후보군을 구성하였다. 응용 후보군 중, 무작위로 Large : Small 비율에 맞게 32, 64개의 응용을 선정해 워크로드를 구성하였다.

본 성능 평가는 워크로드를 구성하는 응용들이 전부 스케줄을 요청한 시간부터 워크로드가 종료되는 시점까지의 시간을 측정하여 선행연구(CASE), Eager Launch(EL), 우선순위를 설정한 Eager Launch(EL_p), 배리어를 설정한 Eager Launch(EL_b), 우선순위 및 배리어를 모두 설정한 Eager Launch(EL_p_b), 두 방법의 단점들을 해결한 Eager Launch(EL_opt)에 따라 성능을 비교함으로써 진행되었다.

그림 4는 각 스케줄링 정책에 따라 발생한 성능을 선행연구에 정규화 시킨 그래프이다. W3과 W4의 경우, 우선순위만 설정한 스케줄러가 배리어만 설정한 스케줄러보다 더 좋은 성능을 나타내는 것을 확인할 수 있다. 그 이유는 워크로드를 구성하는 응용들 중 Normal Launch 태스크에 대한 간섭에 취약한 응용들이 한 GPU에 동시-스케줄되어 실행되는 경우가 없거나 혹은 스케줄 요청 순번이 앞쪽이어서 전체 성능에 있어 성능 저하가 크게 발생하지 않았기 때문이다. 또한, Normal Launch 태스크의 커널 전 작업 소요 시간이 긴 경우, 해당 기간동안 GPU가 idle함에도 불구하고 Eager Launch 태스크는 배리어로 인해 커널 런치가 지연

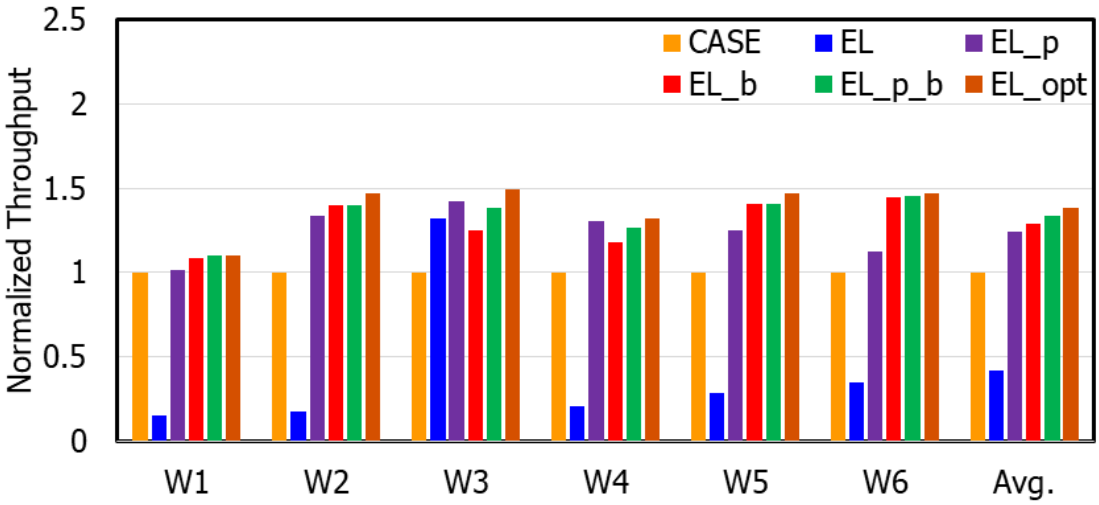

그림 4 스케줄링 정책에 따른 성능 비교

되기 때문에 성능 향상 기회를 놓치게 된다.

반대로 그 외의 워크로드에서는 배리어만 설정해놓은 스케줄러가 우선순위만 설정한 스케줄러보다 더 좋은 성능을 보였는데 이는 Normal Launch 태스크에 대한 Eager Launch 태스크의 자원 간섭이 심하게 발생하여 생긴 성능 저하로 볼 수 있다.

두 방법을 모두 적용한 EL_p_b 스케줄러에서 두 번째로 좋은 성능이 발생하였지만, 해당 스케줄링 정책에서도 여전히 배리어가 Normal Launch 태스크의 모든 커널이 런치가 된 후에 해제된다는 점, GPU가 idle 할 때 Eager Launch 태스크는 불필요하게 런치를 대기하게 된다는 점 등의 문제점이 있어 EL_opt보다 좋은 성능이 나오지 않는 것을 볼 수 있다. 두 문제점을 모두 해결한 EL_opt에서 선행연구 대비 139%로 가장 좋은 성능이 발생하였다.

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/