



(19) 대한민국특허청(KR)
(12) 등록특허공보(B1)

(45) 공고일자 2020년03월09일
(11) 등록번호 10-2086757
(24) 등록일자 2020년03월03일

(51) 국제특허분류(Int. Cl.)
G06F 9/50 (2018.01)
(52) CPC특허분류
G06F 9/5016 (2013.01)
G06F 9/5061 (2013.01)
(21) 출원번호 10-2019-0093262
(22) 출원일자 2019년07월31일
심사청구일자 2019년07월31일
(56) 선행기술조사문헌
KR1020190085391 A*
KR1020190085444 A*
*는 심사관에 의하여 인용된 문헌

(73) 특허권자
서강대학교 산학협력단
서울특별시 마포구 백범로 35 (신수동, 서강대학교)
(72) 발명자
박성용
서울특별시 서대문구 가재울미래로 2, 107동 100
1호(남가좌동, DMC파크뷰자이)
구민우
서울특별시 동작구 사당로17길 52, 12동 1403호(사당동, 대림아파트)
(74) 대리인
유미특허법인

전체 청구항 수 : 총 14 항

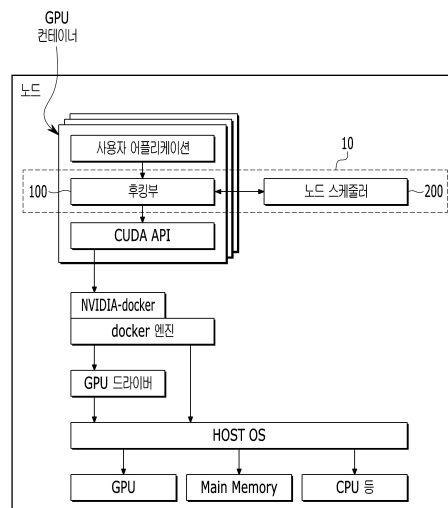
심사관 : 김평수

(54) 발명의 명칭 GPU 메모리 스케줄러 및 이를 이용한 GPU 메모리 선점 방법

(57) 요약

GPU 메모리 스케줄러의 후킹 모듈이 GPU 메모리를 선점하는 방법으로서, 사용자 어플리케이션에서 공유 라이브러리를 호출한 함수가 GPU 메모리 할당 요청 함수이면, 호출 함수를 가로챈다. 가로챈 호출 함수의 변수로부터 GPU 메모리 요청 용량을 확인하고, 확인한 요청 용량을 포함하여 노드 스케줄러로 사용자 어플리케이션으로의 GPU 메모리 할당 가능 여부를 질의한다. 노드 스케줄러로부터 GPU 메모리 할당 승인을 수신하면, 가로챈 호출 함수의 또 다른 변수인 제1 메모리 주소를 제2 메모리 주소로 변경하고, 제2 메모리 주소에서부터 요청 용량만큼 GPU 메모리를 할당 받도록 제2 메모리 주소와 요청 용량을 변수로 하는 GPU 메모리 할당 요청 함수를 공유 라이브러리로 전달한다.

대표도 - 도2



(52) CPC특허분류

G06T 2200/28 (2013.01)

이 발명을 지원한 국가연구개발사업

과제고유번호 1711093225

부처명 과학기술정보통신부

연구관리전문기관 정보통신기획평가원(IITP)

연구사업명 대학ICT연구센터지원사업

연구과제명 지능정보서비스를 위한 고성능 하이브리드 클라우드 컴퓨팅 기술 개발 및 인력양성

기 여 율 1/1

주관기관 건국대학교 산학협력단

연구기간 2019-01-01~2019-12-31

공지예외적용 : 있음

명세서

청구범위

청구항 1

GPU 메모리 스케줄러의 후킹 모듈이 GPU 메모리를 선점하는 방법으로,

사용자 어플리케이션에서 공유 라이브러리를 호출하는 함수를 확인하는 단계,

상기 사용자 어플리케이션이 호출한 호출 함수가 GPU 메모리 할당 요청 함수이면, 상기 호출 함수를 가로채는 (hooking) 단계,

가로챈 상기 호출 함수의 변수로부터 GPU 메모리 요청 용량을 확인하고, 확인한 요청 용량을 포함하여 노드 스케줄러로 상기 사용자 어플리케이션으로의 GPU 메모리 할당 가능 여부를 질의하는 단계, 그리고

상기 노드 스케줄러로부터 GPU 메모리 할당 승인을 수신하면, 상기 가로챈 호출 함수의 또 다른 변수인 제1 메모리 주소를 제2 메모리 주소로 변경하고, 상기 제2 메모리 주소에서부터 요청 용량만큼 GPU 메모리를 할당 받도록 제2 메모리 주소와 요청 용량을 변수로 하는 GPU 메모리 할당 요청 함수를 상기 공유 라이브러리로 전달하는 단계

를 포함하고,

상기 후킹 모듈은 상기 사용자 어플리케이션과 상기 공유 라이브러리와 함께 가상화 환경에 위치하는 복수의 컨테이너에 각각 삽입되어 있는 후킹 모듈들 중 어느 하나이고,

상기 가로챈 호출 함수는 상기 GPU 메모리 할당 요청 함수, 데이터 복사 함수, GPU 메모리 해제 함수, 커널 실행을 위한 라이브러리 설정 함수 그리고 커널 실행 함수 중 어느 하나인 GPU 메모리 선점 방법.

청구항 2

제1항에 있어서,

상기 공유 라이브러리로 전달하는 단계 이후에,

할당 받은 GPU 메모리를 이용한 작업 실행 정지 명령을 상기 노드 스케줄러로부터 수신하는 단계,

상기 GPU 메모리에서 작업하는 데이터를 복사하기 위하여, 상기 GPU 메모리에서 사용하는 용량만큼을 메인 메모리에서 할당 받기 위한 메인 메모리 할당 API를 호출하는 단계,

메인 메모리가 할당되면, 상기 GPU 메모리 상의 데이터를 상기 메인 메모리로 복사하는 단계, 그리고

상기 GPU 메모리의 반환을 상기 노드 스케줄러로 알리는 단계

를 포함하는 GPU 메모리 선점 방법.

청구항 3

제2항에 있어서,

상기 제1 메모리 주소는 상기 사용자 어플리케이션이 상기 GPU 메모리 할당 요청 함수를 처음 호출하였을 때 상기 사용자 어플리케이션에 반환된 메모리 주소이고, 상기 제2 메모리 주소는 상기 데이터가 위치할 메모리의 주소이며,

상기 제2 메모리 주소는 GPU 메모리 주소 또는 메인 메모리 주소 중 어느 하나인 GPU 메모리 선점 방법.

청구항 4

제2항에 있어서,

상기 GPU 메모리의 반환을 상기 노드 스케줄러로 알리는 단계 이후에,

상기 노드 스케줄러로부터 작업 실행 재개 명령을 수신하는 단계,
 상기 GPU 메모리를 할당 받기 위한 GPU 메모리 할당 요청 함수를 호출하는 단계, 그리고
 GPU 메모리가 할당되면, 상기 메인 메모리에 복사되어 있는 상기 데이터를 할당된 GPU 메모리로 복사하는 단계
 를 더 포함하는 GPU 메모리 선점 방법.

청구항 5

삭제

청구항 6

제1항에 있어서,
 상기 GPU 메모리 할당 함수인지 확인하는 단계 이전에,
 상기 노드 스케줄러로 상기 후킹 모듈이 탑재되는 컨테이너를 등록하는 단계, 그리고
 상기 노드 스케줄러로부터 상기 컨테이너의 컨테이너 식별 정보를 수신하는 단계
 를 더 포함하는 GPU 메모리 선점 방법.

청구항 7

GPU 메모리 스케줄러의 노드 스케줄러가 GPU 메모리를 선점하는 방법으로서,
 상기 GPU 메모리의 일부를 제1 GPU 컨테이너의 제1 사용자 어플리케이션에서 사용하고 있으며,
 제2 GPU 컨테이너의 후킹부로부터, 상기 제2 GPU 컨테이너에 포함된 제2 사용자 어플리케이션이 요청한 GPU 메모리의 할당 가능 질의를 수신하는 단계,
 상기 GPU 메모리의 가용 용량이 상기 제2 사용자 어플리케이션에서 요청한 GPU 메모리 요청 용량보다 적으면,
 상기 제1 GPU 컨테이너의 후킹부로 상기 GPU 메모리에서의 작업 실행 정지 명령을 전송하는 단계,
 상기 제1 GPU 컨테이너의 후킹부로부터 GPU 메모리 반환 알림을 수신하면, 상기 제2 GPU 컨테이너의 후킹부로 메모리 할당 허가 신호를 전송하는 단계, 그리고
 상기 제1 GPU 컨테이너와 제2 GPU 컨테이너의 정보를 포함하는 컨테이너 리스트 테이블을 갱신하는 단계
 를 포함하고,
 상기 GPU 메모리 스케줄러는 하나의 상기 노드 스케줄러와 복수의 GPU 컨테이너들 각각에 삽입된 후킹부들을 포함하며,
 상기 제1 GPU 컨테이너는 상기 제1 사용자 어플리케이션과 상기 제1 GPU 컨테이너의 후킹부를 포함하여 가상화 환경에 위치하고, 상기 제2 GPU 컨테이너는 상기 제2 사용자 어플리케이션과 상기 제2 GPU 컨테이너의 후킹부를 포함하여 가상화 환경에 위치하는 GPU 메모리 선점 방법.

청구항 8

제7항에 있어서,
 상기 작업 실행 정지 명령을 전송하는 단계는,
 상기 제1 사용자 어플리케이션이 상기 GPU 메모리에서 처리하는 작업과 상기 제2 사용자 어플리케이션이 상기 GPU 메모리에서 처리할 작업의 우선 순위를 비교하는 단계, 그리고
 상기 처리할 작업의 우선 순위가 높으면, 상기 제1 GPU 컨테이너의 후킹부로 상기 실행 정지 명령을 전송하는 단계
 를 포함하는 GPU 메모리 선점 방법.

청구항 9

제7항에 있어서,

상기 컨테이너 리스트 테이블은 상기 제1 GPU 컨테이너와 제2 GPU 컨테이너 각각의 컨테이너 식별 정보, 컨테이너들 각각의 사용자 어플리케이션이 GPU 메모리 할당을 시도한 총 메모리 용량, 현재 GPU 메모리를 할당 받아 사용하고 있는 메모리 용량, 그리고 상기 GPU 메모리에서 작업이 실행중인지 여부를 나타내는 상태 정보를 포함하는 GPU 메모리 선점 방법.

청구항 10

대상 사용자 어플리케이션에서 물리적인 GPU 메모리를 사용하기 위해 공유 라이브러리를 호출하는 호출 함수를 가로채고, 가로챈 호출 함수에 따라 상기 대상 사용자 어플리케이션에서 출력된 데이터가 향할 메모리 주소를 변경하여 상기 공유 라이브러리로 호출 함수를 전달하는 후킹부, 그리고

상기 후킹부에서 전달되는 호출 함수들 중 GPU 메모리 할당 가능 질의를 수신하면, 상기 대상 사용자 어플리케이션이 상기 물리적인 GPU 메모리를 선점하도록 상기 물리적인 GPU 메모리를 공유하는 복수의 사용자 어플리케이션들 중 GPU 메모리 사용 정지 명령 대상 사용자 어플리케이션을 선택하고, 선택된 사용 정지 명령 대상 사용자 어플리케이션과 연결되어 있는 후킹부로 사용 정지 명령을 전송하는 노드 스케줄러

를 포함하고,

상기 후킹부는 상기 사용자 어플리케이션 및 공유 라이브러리와 함께 컨테이너에 포함되어 가상화 환경에 각각 위치하고, 상기 노드 스케줄러는 가상화 환경에 위치한 각 후킹부들과 연동하는 GPU 메모리 스케줄러.

청구항 11

제10항에 있어서,

상기 후킹부는,

상기 공유 라이브러리를 호출하는 함수가 GPU 메모리 할당을 위한 함수, GPU 메모리 해제를 위한 함수, GPU 메모리와 메인 메모리 사이에 데이터 복사를 위한 함수, 그리고 커널 실행 관련 호출 함수 중 어느 하나이면, 상기 공유 라이브러리로 향하는 호출 함수를 가로채는 GPU 메모리 스케줄러.

청구항 12

제11항에 있어서,

상기 후킹부는,

상기 노드 스케줄러로부터 사용 정지 명령을 수신하면, 상기 GPU 메모리에서 작업중인 데이터를 메인 메모리로 복사하기 위하여, 상기 GPU 메모리에서 사용하는 용량만큼의 메인 메모리 할당 함수를 호출하는 GPU 메모리 스케줄러.

청구항 13

제12항에 있어서,

상기 후킹부는,

상기 호출 함수에 포함된 사용자 메모리 주소(ptrUser)를 실제 메모리 주소(ptrReal)로 변환하고,

상기 사용자 메모리 주소는 상기 GPU 메모리 할당 요청 함수가 처음 호출될 때 상기 대상 사용자 어플리케이션에 반환된 메모리 주소이고, 상기 실제 메모리 주소는 상기 데이터가 위치한 메모리의 메모리 주소이며,

상기 실제 메모리 주소는 GPU 메모리 주소 또는 메인 메모리 주소 중 어느 하나인 GPU 메모리 스케줄러.

청구항 14

제13항에 있어서,

상기 노드 스케줄러는,

상기 대상 사용자 어플리케이션이 상기 GPU 메모리에서 처리할 작업과 상기 GPU 메모리를 공유하고 있는 복수의

사용자 어플리케이션들 각각의 작업의 우선순위를 비교하고, 비교한 우선순위에 따라 상기 사용 정지 명령을 수신할 사용자 어플리케이션을 결정하는 GPU 메모리 스케줄러.

청구항 15

제14항에 있어서,

상기 노드 스케줄러는,

상기 GPU 메모리를 공유하거나 공유하고자 하는 복수의 컨테이너들 각각의 컨테이너 식별자를 생성하여 상기 후킹부로 전달하고,

상기 복수의 컨테이너들 각각에 대한 컨테이너 식별자, 각각의 컨테이너들에서 GPU 메모리 할당을 시도한 메모리 용량, 현재 GPU 메모리를 할당 받아 사용하고 있는 메모리 용량, 상기 GPU 메모리에서 작업이 실행중인지 여부를 나타내는 상태 정보를 포함하는 컨테이너 리스트 테이블을 관리하는 GPU 메모리 스케줄러.

발명의 설명

기술 분야

[0001] 본 발명은 GPU 메모리 스케줄러 및 이를 이용한 GPU 메모리 선점 방법에 관한 것이다.

배경 기술

[0002] GPGPU(General-Purpose computing on Graphics Processing Units) 등 GPU 기술의 발전으로 그래픽스뿐만 아니라 다양한 용도로 GPU를 활용할 수 있게 되었고, 이로 인해 다양한 분야의 연산에 GPU가 사용되고 있다. GPU는 비싼 자원이므로, GPU의 사용률을 높이기 위해 워크로드 간에 GPU를 공유하기 시작하였다.

[0003] GPU를 공유하는 환경에서는 긴급한 작업을 위해 다른 작업에 우선하여 GPU의 선점이 필요한 경우가 발생하는데, GPU 메모리를 선점하는 것이 불가능했다. 이에 종래에는 GPU를 아예 선점할 수 없도록 하는 방식, GPU 메모리는 선점하지 못하나 연산만을 선점하도록 하는 방식, 또는 GPU 메모리에서 수행되고 있는 기존 작업을 강제로 종료하여 선점하는 강제 종료 기반 선점 방식(Kill-based preemption) 중 하나의 방식을 사용하였다.

[0004] 이 중 강제 종료 기반 선점 방식은 GPU 메모리를 선점하지 못하는 상황에서 긴급한 작업의 우선 실행에 필요한 메모리를 확보하기 위하여, 기존 작업을 강제로 종료하는 방법이다. 기존 작업을 강제로 종료하면 GPU의 가용 메모리는 즉시 확보할 수 있지만, GPU에서 작업 중이던 기존 작업의 상태가 저장되지 않기 때문에, 작업 내용의 손실이 유발되어 오버헤드가 매우 크다.

[0005] 이 외에도 상위 라이브러리 즉, 특정 딥러닝(deep learning) 라이브러리를 수정하여 딥러닝을 기반으로 GPU 메모리를 선점하는 방식도 있다. 딥러닝 기반의 GPU 메모리 선점 기능을 사용하기 위해서는, GPU 메모리를 공유하여 사용하는 모든 작업이 특정 딥러닝 라이브러리를 사용해야 하기 때문에, 라이브러리 종속성의 문제가 있다.

발명의 내용

해결하려는 과제

[0006] 따라서, 본 발명은 다양한 종류의 응용 프로그램이 GPU 메모리를 공유하는 환경에서, 사용자 어플리케이션의 소스 코드를 수정하거나 기존 작업을 강제 종료하지 않고도 GPU 메모리를 선점할 수 있도록 스케줄링할 수 있는 GPU 메모리 스케줄러 및 이를 이용한 GPU 메모리 선점과 스케줄링 방법을 제공한다.

과제의 해결 수단

[0007] 상기 본 발명의 기술적 과제를 달성하기 위한 본 발명의 하나의 특징인 GPU 메모리 스케줄러의 후킹 모듈이 GPU 메모리를 선점하는 방법으로서,

[0008] 사용자 어플리케이션에서 공유 라이브러리를 호출하는 함수를 확인하는 단계, 상기 사용자 어플리케이션이 호출한 호출 함수가 GPU 메모리 할당 요청 함수이면, 상기 호출 함수를 가로채는(hooking) 단계, 가로챈 상기 호출 함수의 변수로부터 GPU 메모리 요청 용량을 확인하고, 확인한 요청 용량을 포함하여 노드 스케줄러로 상기 사용자 어플리케이션으로의 GPU 메모리 할당 가능 여부를 질의하는 단계, 그리고 상기 노드 스케줄러로부터 GPU 메

메모리 용량 할당 승인을 수신하면, 상기 가로챌 호출 함수의 또 다른 변수인 제1 메모리 주소를 제2 메모리 주소로 변경하고, 상기 제2 메모리 주소에서부터 요청 용량만큼 GPU 메모리를 할당 받도록 제2 메모리 주소와 요청 용량을 변수로 하는 GPU 메모리 할당 요청 함수를 상기 공유 라이브러리로 전달하는 단계를 포함한다.

- [0009] 상기 공유 라이브러리로 전달하는 단계 이후에,
- [0010] 할당 받은 GPU 메모리를 이용한 작업 실행 정지 명령을 상기 노드 스케줄러로부터 수신하는 단계, 상기 GPU 메모리에서 작업하는 데이터를 복사하기 위하여, 상기 GPU 메모리에서 사용하는 용량만큼을 메인 메모리에서 할당 받기 위한 메인 메모리 할당 API를 호출하는 단계, 메인 메모리가 할당되면, 상기 GPU 메모리 상의 데이터를 상기 메인 메모리로 복사하는 단계, 그리고 상기 GPU 메모리의 반환을 상기 노드 스케줄러로 알리는 단계를 포함할 수 있다.
- [0011] 상기 제1 메모리 주소는 상기 사용자 어플리케이션이 상기 GPU 메모리 할당 요청 함수를 처음 호출하였을 때 상기 사용자 어플리케이션에 반환된 메모리 주소이고, 상기 제2 메모리 주소는 상기 데이터가 위치할 메모리의 주소이며, 상기 제2 메모리 주소는 GPU 메모리 주소 또는 메인 메모리 주소 중 어느 하나일 수 있다.
- [0012] 상기 GPU 메모리의 반환을 상기 노드 스케줄러로 알리는 단계 이후에,
- [0013] 상기 노드 스케줄러로부터 작업 실행 재개 명령을 수신하는 단계, 상기 GPU 메모리를 할당 받기 위한 GPU 메모리 할당 요청 함수를 호출하는 단계, 그리고 GPU 메모리가 할당되면, 상기 메인 메모리에 복사되어 있는 상기 데이터를 할당된 GPU 메모리로 복사하는 단계를 포함할 수 있다.
- [0014] 상기 가로챌 호출 함수는 상기 GPU 메모리 할당 요청 함수, 데이터 복사 함수, GPU 메모리 해제 함수, 커널 실행을 위한 라이브러리 설정 함수 그리고 커널 실행 함수 중 어느 하나일 수 있다.
- [0015] 상기 GPU 메모리 할당 함수인지 확인하는 단계 이전에, 상기 노드 스케줄러로 상기 컨테이너를 등록하는 단계, 그리고 상기 노드 스케줄러로부터 상기 컨테이너의 컨테이너 식별 정보를 수신하는 단계를 더 포함할 수 있다.
- [0016] 상기 본 발명의 기술적 과제를 달성하기 위한 본 발명의 또 다른 특징인 GPU 메모리 스케줄러의 노드 스케줄러가 GPU 메모리를 선점하는 방법으로서,
- [0017] 상기 GPU 메모리의 일부를 제1 GPU 컨테이너의 제1 사용자 어플리케이션에서 사용하고 있으며, 제2 GPU 컨테이너의 후킹부로부터, 상기 제2 GPU 컨테이너에 포함된 제2 사용자 어플리케이션이 요청한 GPU 메모리의 할당 가능 질의를 수신하는 단계, 상기 GPU 메모리의 가용 용량이 상기 제2 사용자 어플리케이션에서 요청한 GPU 메모리 요청 용량보다 적으면, 상기 제1 GPU 컨테이너의 후킹부로 상기 GPU 메모리에서의 작업 실행 정지 명령을 전송하는 단계, 상기 제1 GPU 컨테이너의 후킹부로부터 GPU 메모리 반환 알림을 수신하면, 상기 제2 GPU 컨테이너의 후킹부로 메모리 할당 허가 신호를 전송하는 단계, 그리고 상기 제1 GPU 컨테이너와 제2 GPU 컨테이너의 정보를 포함하는 컨테이너 리스트 테이블을 갱신하는 단계를 포함하고, 상기 GPU 메모리 스케줄러는 하나의 상기 노드 스케줄러와 복수의 GPU 컨테이너들 각각에 삽입된 후킹부들을 포함한다.
- [0018] 상기 작업 실행 정지 명령을 전송하는 단계는, 상기 제1 사용자 어플리케이션이 상기 GPU 메모리에서 처리하는 작업과 상기 제2 사용자 어플리케이션이 상기 GPU 메모리에서 처리할 작업의 우선 순위를 비교하는 단계, 그리고 상기 처리할 작업의 우선 순위가 높으면, 상기 제1 GPU 컨테이너의 후킹부로 상기 실행 정지 명령을 전송하는 단계를 포함할 수 있다.
- [0019] 상기 컨테이너 리스트 테이블은 상기 제1 GPU 컨테이너와 제2 GPU 컨테이너 각각의 컨테이너 식별 정보, 컨테이너들 각각의 사용자 어플리케이션이 GPU 메모리 할당을 시도한 총 메모리 용량, 현재 GPU 메모리를 할당 받아 사용하고 있는 메모리 용량, 그리고 상기 GPU 메모리에서 작업이 실행중인지 여부를 나타내는 상태 정보를 포함할 수 있다.
- [0020] 상기 본 발명의 기술적 과제를 달성하기 위한 본 발명의 또 다른 특징인 GPU 메모리 스케줄러는, 대상 사용자 어플리케이션에서 GPU 메모리를 사용하기 위해 공유 라이브러리를 호출하는 호출 함수를 가로채고, 가로챌 호출 함수에 따라 데이터가 향할 메모리 주소를 변경하여 상기 공유 라이브러리로 호출 함수를 전달하는 후킹부, 그리고 상기 후킹부에서 전달되는 호출 함수들 중 GPU 메모리 할당 가능 질의를 수신하면, 상기 대상 사용자 어플리케이션이 상기 GPU 메모리를 선점하도록 상기 GPU 메모리를 공유하는 복수의 사용자 어플리케이션들 중 GPU 메모리 사용 정지 명령 대상 사용자 어플리케이션을 선택하고, 선택된 사용 정지 명령 대상 사용자 어플리케이션과 연결되어 있는 후킹부로 사용 정지 명령을 전송하는 노드 스케줄러를 포함하고, 상기 후킹부는 상기 사용자 어플리케이션과 공유 라이브러리와 함께 컨테이너에 포함되어 가상화 환경에 위치하고, 상기 노드 스케줄러

는 복수의 후킹부들과 연동한다.

- [0021] 상기 후킹부는, 상기 공유 라이브러리를 호출하는 함수가 GPU 메모리 할당을 위한 함수, GPU 메모리 해제를 위한 함수, GPU 메모리와 메인 메모리 사이에 데이터 복사를 위한 함수, 그리고 커널 실행 관련 호출 함수 중 어느 하나이면, 상기 공유 라이브러리로 향하는 호출 함수를 가로챌 수 있다.
- [0022] 상기 후킹부는, 상기 노드 스케줄러로부터 사용 정지 명령을 수신하면, 상기 GPU 메모리에서 작업중인 데이터를 메인 메모리로 복사하기 위하여, 상기 GPU 메모리에서 사용하는 용량만큼의 메인 메모리 할당 함수를 호출할 수 있다.
- [0023] 상기 후킹부는, 상기 호출 함수에 포함된 사용자 메모리 주소(ptrUser)를 실제 메모리 주소(ptrReal)로 변환하고, 상기 사용자 메모리 주소는 상기 GPU 메모리 할당 요청 함수가 처음 호출될 때 상기 대상 사용자 어플리케이션에 반환된 메모리 주소이고, 상기 실제 메모리 주소는 상기 데이터가 위치한 메모리의 메모리 주소이며, 상기 실제 메모리 주소는 GPU 메모리 주소 또는 메인 메모리 주소 중 어느 하나일 수 있다.
- [0024] 상기 노드 스케줄러는, 상기 대상 사용자 어플리케이션이 상기 GPU 메모리에서 처리할 작업과 상기 GPU 메모리를 공유하고 있는 복수의 사용자 어플리케이션들 각각의 작업의 우선순위를 비교하고, 비교한 우선순위에 따라 상기 사용 정지 명령을 수신할 사용자 어플리케이션을 결정할 수 있다.
- [0025] 상기 노드 스케줄러는, 상기 GPU 메모리를 공유하거나 공유하고자 하는 복수의 컨테이너들 각각의 컨테이너 식별자를 생성하여 상기 후킹부로 전달하고, 상기 복수의 컨테이너들 각각에 대한 컨테이너 식별자, 각각의 컨테이너들에서 GPU 메모리 할당을 시도한 메모리 용량, 현재 GPU 메모리를 할당 받아 사용하고 있는 메모리 용량, 상기 GPU 메모리에서 작업이 실행중인지 여부를 나타내는 상태 정보를 포함하는 컨테이너 리스트 테이블을 관리할 수 있다.

발명의 효과

- [0026] 본 발명에 따르면, 다른 작업에 할당되어 GPU 메모리 용량이 부족한 상황에서도, 긴급한 작업에 대해 GPU 메모리 선점을 지원할 수 있다.
- [0027] 또한, 본 발명에 따르면, GPU 메모리를 공유하는 작업들의 GPU 메모리 사용량이 GPU 메모리의 물리적 용량의 크기보다 큰 경우에도 강제 종료 없이 실행시킬 수 있다.

도면의 간단한 설명

- [0028] 도 1은 일반적인 CUDA API 구조에 대한 예시도이다.
- 도 2는 본 발명의 실시예에 따른 GPU 메모리 스케줄러의 구조도이다.
- 도 3은 본 발명의 실시예에 따른 후킹부와 노드 스케줄러의 구조도이다.
- 도 4는 본 발명의 실시예에 따른 후킹부의 초기 절차에 대한 흐름도이다.
- 도 5는 본 발명의 실시예에 따른 GPU 메모리 선점 방법에 대한 흐름도이다.
- 도 6은 본 발명의 실시예에 따른 사용자 메모리 주소 충돌을 나타낸 예시도이다.
- 도 7은 본 발명의 실시예에 따른 오브젝트 단위 GPU 메모리 스왑에 대한 예시도이다.
- 도 8은 본 발명의 실시예에 따른 패킹 스케줄링 정책의 예시도이다.
- 도 9는 본 발명의 실시예에 따라 시간에 따른 GPU 메모리 사용량 및 GPU 사용률 변화를 나타낸 그래프이다.

발명을 실시하기 위한 구체적인 내용

- [0029] 아래에서는 첨부한 도면을 참고로 하여 본 발명의 실시예에 대하여 본 발명이 속하는 기술 분야에서 통상의 지식을 가진 자가 용이하게 실시할 수 있도록 상세히 설명한다. 그러나 본 발명은 여러 가지 상이한 형태로 구현될 수 있으며 여기에서 설명하는 실시예에 한정되지 않는다. 그리고 도면에서 본 발명을 명확하게 설명하기 위해서 설명과 관계없는 부분은 생략하였으며, 명세서 전체를 통하여 유사한 부분에 대해서는 유사한 도면 부호를 붙였다.
- [0030] 명세서 전체에서, 어떤 부분이 어떤 구성요소를 "포함"한다고 할 때, 이는 특별히 반대되는 기재가 없는 한 다

른 구성요소를 제외하는 것이 아니라 다른 구성요소를 더 포함할 수 있는 것을 의미한다.

- [0031] 또한, 명세서 전체에서, 특별히 "메인 메모리"라고 명시하지 않는 한, "메모리"는 메인 메모리가 아닌 "GPU 메모리"를 의미하며, "GPU 메모리"는 별다른 보충 설명이 없는 한 "GPU 전역 메모리(GPU Global Memory)"를 의미한다.
- [0032] 이하 도면을 참조로 하여 본 발명의 실시예에 따른 GPU 메모리 스케줄러 및 이를 이용한 GPU 메모리 선점 방법에 대해 설명한다.
- [0033] 본 발명의 실시예에 대해 설명하기 앞서, 일반적인 CUDA API(Compute Unified Device Architecture Application Program Interface) 구조에 대해 도 1을 참조로 설명한다. CUDA는 NVIDIA가 개발한 병렬 컴퓨팅 서비스 플랫폼 및 API 모델로, 소프트웨어 개발자와 소프트웨어 엔지니어가 범용 프로세싱을 위해 CUDA 지원 GPU를 사용한다.
- [0034] 도 1은 일반적인 CUDA API 구조에 대한 예시도이다.
- [0035] 도 1의 (a)에 나타낸 바와 같이 CUDA는 CUDA Driver API(이하, '드라이버 API' 라 지칭함)와 CUDA Runtime API(이하, '런타임 API' 라 지칭함)를 제공한다. 그리고 도면에 도시하지 않았으나 CUDA Math API 등의 여러 API들을 지원한다.
- [0036] CUDA API는 CUDA 기본 선형 대수학 서브 루틴 라이브러리인 cuBLAS, CUDA 고속 푸리에 변환 라이브러리인 cuFFT 등의 라이브러리들을 제공한다. 이 라이브러리들은 런타임 API나 드라이버 API의 위에서 구현되어 있다.
- [0037] 드라이버 API는 저 수준(low-level)의 API로, 사용자로 하여금 구현의 편리성은 다소 떨어진다. 그러나 GPU 장치의 초기화, 컨텍스트 관리, 모듈 관리 등 GPU 장치의 제어를 사용자가 모두 조절할 수 있어 자유도가 높다. 런타임 API는 드라이버 API 상에 구현된 고급 API로, 사용자로 하여금 CUDA의 기능을 편리하게 사용하도록 구현된 것이다.
- [0038] 이때, 도 1의 (b)에 도시된 바와 같이, 사용자 어플리케이션과 공유 라이브러리인 CUDA API는 도커(docker) 컨테이너 기반으로 가상화 환경에서 구현될 수도 있다. 하나의 도커에 연결되어 있는 복수의 GPU 컨테이너들 각각은, 사용자 어플리케이션과 CUDA API 로 이루어진다.
- [0039] GPU를 이용하면서 여러 환경의 CUDA 툴킷을 이용할 경우, 도커라는 확장 기능을 추가하면 편리하게 사용할 수는 있다. 그러나 기본적으로 도커를 통해서 컨테이너를 실행하면, GPU의 자원은 사용할 수 없다.
- [0040] 한편, GPU의 마이크로-아키텍처는 인스트럭션 단위(instruction-level granularity)로 선점 기능(Compute Preemption)을 지원한다. 이는 연산을 선점하거나, 커널(kernel)의 실행 도중에 레지스터나 공유 메모리는 선점할 수 있으나, GPU 전역 메모리(global memory)를 선점하지는 못한다. 그리고 선점 기능은 하드웨어적으로 자동으로 작동하기 때문에, 개발자가 응용 프로그램이나 운영체제 수준에서 소프트웨어적으로는 조절이 불가능하다.
- [0041] 또한, GPU는 메모리 스왑(swap) 기능을 제공하지 않는다. 따라서 가상화 환경에서 각 가상 머신 또는 컨테이너가 할당 받을 수 있는 GPU 메모리 양의 합계는, 물리적인 GPU 메모리 크기를 초과하지 못한다.
- [0042] GPU에서 CUDA 응용 프로그램을 실행할 때, 커널의 실행을 정지할 방법은 현재까지 없다. CUDA API에서 제공하는 cudaDeviceReset() 기능을 사용하면 커널의 실행을 중단할 수는 있다. 그러나 해당 GPU가 완전히 초기화되기 때문에, GPU를 사용하는 모든 응용 프로그램의 작업 상태가 손실된다.
- [0043] GPU 메모리를 공유하거나, 더 나아가 GPU 메모리를 선점하려 시도했던 다양한 연구들이 있었다. 예를 들어, GPU 메모리 선점에 대한 기능이 제공되지 않기 때문에, 강제 종료 기반 선점 방식을 사용하거나, GPU 관련 응용 프로그램 인터페이스 호출을 가로채 지연시키는 방식으로 GPU 메모리를 선점하거나, 또는 GPU 관련 응용 프로그램 인터페이스보다 더 상위에 있는 프레임워크를 통해 간접적으로 GPU 메모리 선점을 지원하고 있다.
- [0044] 그러나 GPU 메모리에서 처리 중인 작업을 중단하거나 GPU 메모리를 회수할 방법이 없어, 단순히 실행 중인 컨테이너나 프로세스를 강제 종료하여 자원을 회수하는 방식인 강제 종료 기반 선점 방식이 사용되고 있다. 강제 종료 기반 선점 방식은 기존 작업의 상태가 저장되지 않고 강제 종료되기 때문에, 사용자가 중간에 수동으로 저장하지 않은 작업 내용은 모두 손실된다. 손실된 작업은 다시 수행해야 하므로 오버헤드가 매우 크다.
- [0045] 이 외에도 GPU 메모리를 공유하는 가상 환경에서, GPU를 스케줄링하는 기법들이 연구되었다. 그러나, 종래 기법들을 이용한다 하더라도 GPU 메모리 선점 기능을 제공하지 않기 때문에, 메모리가 가득 찼을 때 새로 시작되었

거나 GPU 메모리를 추가로 필요로 하는 작업들은, 여전히 가용 메모리가 확보될 때까지 멈춰서 기다릴 수밖에 없다. 따라서 이들은 가용 메모리가 부족한 경우에는 긴급한 작업이 들어와도 실행할 수 없다.

[0046] 따라서, 본 발명의 실시예에서는 플랫폼 종속성이 없고, GPU 컨텍스트 전환 기능을 갖춘 GPU 선점형 스케줄러와 이를 이용한 노드 스케줄링 방법에 대해 제안한다.

[0047] 도 2는 본 발명의 실시예에 따른 GPU 메모리 스케줄러의 구조도이다.

[0048] 도 2에 도시된 바와 같이, 본 발명의 실시예에 따른 GPU 메모리 스케줄러(10)는 종래의 CUDA API 구조에 후킹부(100)와 노드 스케줄러(200)를 삽입한 형태로 구현함으로써, 플랫폼 종속성 없이 GPU 컨텍스트 전환 기능을 제공할 수 있다. 본 발명의 실시예에서는 설명의 편의를 위하여 후킹부(100)와 노드 스케줄러(200)를 포함하여 ‘GPU 메모리 스케줄러(10)’라 지칭한다.

[0049] GPU 메모리 스케줄러(10)의 후킹부(100)는 GPU 컨테이너에 삽입되어 있고, 노드 스케줄러(200)는 노드 내에 삽입되어 있다. 여기서 노드라 함은 PC를 의미한다. 후킹부(100)는 GPU 컨테이너의 수만큼 생성되며, 노드 스케줄러(200)는 노드 내에 하나만 삽입되어 있다.

[0050] 후킹부(100)는 GPU 컨테이너가 생성될 때 베이스 이미지에 선 탑재되어 생성된다. 그리고 사용자 어플리케이션에서 CUDA API를 호출하지 않으면 비활성 상태를 유지하다가, CUDA API를 호출함과 동시에 활성 상태로 변경된다.

[0051] 후킹부(100)는 LD_PRELOAD 환경 변수 값으로 지정된다. 후킹부(100)는 사용자 어플리케이션에서 호출하는 복수의 API 호출들 중 CUDA API 호출(메모리 관련 호출, 커널 실행 관련 호출)이 발생하면, 해당 호출을 가로챈다. 여기서 LD_PRELOAD는 리눅스 시스템에서 사용되는 환경 변수의 하나로, 공유 오브젝트 파일의 경로를 설정할 수 있다. 즉, LD_PRELOAD에 설정한 공유 오브젝트는 다른 모든 공유 오브젝트보다 먼저 로드(load)된다.

[0052] 따라서, 기존 공유 라이브러리와 함수명(symbol)이 동일한 함수가 LD_PRELOAD 설정된 공유 오브젝트에 정의되어 있다고 가정한다. 그러면, 해당 이름의 함수가 호출될 경우 기존 공유 라이브러리에 있는 함수가 아닌, LD_PRELOAD 설정된 공유 오브젝트에 있는 동일한 이름의 함수가 호출된다. 이 기법을 사용하면 공유 라이브러리로의 호출을 가로챌 수 있다

[0053] 또한, 후킹부(100)는 사용자 어플리케이션(혹은, 사용자 프로그램이라고도 지칭함)에서 호출된 GPU 메모리 할당 요청이나 해제 요청, 커널 실행 요청 등을 관리하며, 메모리 주소 변환 등 사용자 어플리케이션에 직접 영향을 줄 수 있는 부분들을 처리한다. 이때, GPU 메모리 할당 요청을 받은 후킹부(100)는 이미 할당되어 있는 GPU 메모리의 주소 영역에 중복되지 않도록 주소를 할당하여 사용자 어플리케이션에 반환한다.

[0054] 후킹부(100)는 통신을 통해 노드 스케줄러(200)로 GPU 메모리 할당이나 해제 등 GPU 메모리 상태를 보고하고, GPU 메모리의 신규 할당이 가능한지 승인을 요청한다. 그리고 노드 스케줄러(200)로부터 GPU 메모리 사용 정지 제어 신호, GPU 메모리 신규 할당 허가 신호 등의 신호들과, 노드 스케줄러(200)가 생성한 컨테이너 식별 정보를 수신하기도 한다.

[0055] 노드 스케줄러(200)는 노드에 하나만 존재한다. 노드 스케줄러(200)는 현재 노드에서 실행되고 있는 컨테이너들의 메모리 할당 정보와 컨테이너들이 실행 함수를 호출한 커널 실행 등을 감시한다. 그리고 노드 스케줄러(200)는 후킹부(100)에서 전송되는 GPU 메모리 할당 가능 질의나 커널 실행 가능 질의를 검토하여, GPU 메모리 승인/커널 실행을 승인하거나 거부하기도 한다.

[0056] 노드 스케줄러(200)는 메모리 스케줄링 알고리즘을 적용하여, 필요 시 GPU 메모리의 선점이나 GPU 메모리 사용 중단 등의 명령을 후킹부(100)로 전송한다. 후킹부(100)와 노드 스케줄러(200)는 원격 프로시저 호출을 사용해 통신한다.

[0057] 후킹부(100)와 노드 스케줄러(200)의 구조에 대해 도 3을 참조로 상세히 설명한다. 후킹부(100)는 C 언어로 구현되어 공유 오브젝트로 컴파일되나, 본 발명의 실시예에서는 설명의 편의를 위하여 노드 스케줄러(200)와 같이 하드웨어 형태로 구현된 것을 예로 하여 설명한다.

[0058] 도 3은 본 발명의 실시예에 따른 후킹부와 노드 스케줄러의 구조도이다.

[0059] 도 3에 도시된 바와 같이 후킹부(100)와 노드 스케줄러(200)는 각각 통신 모듈을 통해, 원격 프로시저를 호출하여 상호 통신을 수행한다.

[0060] 먼저 후킹부(100)는 후킹 모듈(110), 메모리 변환 테이블(120), 통신 모듈(130)을 포함한다.

[0061] 후킹 모듈(110)은 처음 컨테이너가 생성될 때 비활성 상태로 함께 생성된다. 후킹 모듈(110)은 사용자 어플리케이션에서 전송되는 복수의 API 호출들 중 사용자 어플리케이션의 CUDA API가 호출되면 후킹부(100)와 함께 활성 상태가 된다. 그리고 CUDA API 호출 중 메모리 관련 호출과 커널 실행 관련 호출들만 가로챈다. 여기서 후킹 모듈(110)이 가로챈 메모리 관련 호출과 커널 실행 관련 호출(이하, 설명의 편의를 위하여, ‘후킹 대상 API 호출’이라 지칭함)은 다음 표 1과 같다.

표 1

메모리 관련 호출	cudaMalloc
	cudaMemcpy
	cudaFree
커널 실행 관련 호출	cudaSetupArgument
	cudaLaunch

[0063] cudaMalloc은 GPU 메모리를 할당하기 위한 함수이고, cudaFree는 사용하고 있는 GPU 메모리를 해제하기 위한 함수이다. 그리고 cudaMemcpy는 GPU 메모리에서 메인 메모리로 데이터를 복사하거나, 반대로 메인 메모리에서 GPU 메모리로 데이터를 복사할 때 사용되는 함수이다.

[0064] 그리고 cudaSetupArgument는 커널 실행에 필요한 인자들을 설정하는 함수이고, cudaLaunch는 cudaSetupArgument에서 설정된 인자들을 사용하는 커널을 실제로 실행하는 함수이다. 각각의 함수의 기능은 이미 알려진 것으로 본 발명의 실시예에서는 상세한 설명은 생략한다.

[0065] 후킹 모듈(110)은 위에서 언급한 함수를 가로채고 사용자 어플리케이션에 필요한 GPU 메모리를 선점하기 위하여, 다음 표 2의 GPU 메모리 선점 코드가 삽입되어 있다.

표 2

[0066]

```

110. float *A, *B;
120. cudaMalloc(&A, size);; cudaMalloc(&B, size);
130. cudaMemcpy(A, data, size, HostToDevice);
140. kernel<<<X, Y>>>(A, B, size);
210. tmpB-malloc(size);
220. cudaMemcpy(tmpB, B, size, cudaMemcpyDeviceToHost);
230. cudaFree(B);
310. /* other job - waiting */
240. cudaMalloc(&B, size);
250. cudaMemcpy(B, tmpB, size, cudaMemcpyHostToDevice);
260. free(tmpB);
150. kernel2<<<X, Y>>>(B, size);
160. cudaMemcpy(result, B, size, cudaMemcpyDeviceToHost);
170. cudaFree(A); cudaFree(B);
    
```

[0067] 표 2에서 볼드체로 표현한 210~260번 코드, 그리고 310번의 코드들이 새로 삽입된 코드들이다.

[0068] 표 2를 설명하면, 두 개의 컨테이너(A, B)가 GPU 메모리를 할당 받아 작업을 수행하고 있다고 가정한다. 그리고 140번 코드가 실행된 시점에 또 다른 컨테이너에 의해 GPU 메모리 선점이 요구되어, 두 개의 컨테이너 중 B 컨테이너가 노드 스케줄러(200)로부터 실행 중지 명령을 받았다고 가정한다.

[0069] 그러면, 컨테이너 B에 삽입된 후킹 모듈(110)은 메모리 할당 관련 정보를 관리하는 테이블인 메모리 변환 테이블(120)에서 메모리 할당에 관련된 정보를 읽어 들인다. 이는 컨테이너 B의 사용자 어플리케이션이 GPU 메모리에서 할당 받아 사용하던 메모리 크기와 동일하게, 메인 메모리를 할당 받아야 하기 때문이다(210번 코드),

[0070] GPU 메모리-메인 메모리(Device-to-Host) 방향의 cudaMemcpy() 함수 호출로, GPU에 있는 전역 메모리에서 처리되고 있는 데이터를 메인 메모리로 스왑-아웃(swap out)한다(220번 코드). 그 후 컨테이너 B의 후킹 모듈(110)

은 `cudaFree()`를 호출하여, 자신이 사용하던 해당 GPU 메모리의 할당을 해제한다(230번 코드). 그리고 GPU 메모리를 이용하여 처리하던 작업을 별도의 지시가 있을 때까지 대기한다(310번 코드).

- [0071] 또 다른 컨테이너가 선점의 목적을 달성하여 GPU 메모리를 해제하게 되면, 노드 스케줄러(200)는 컨테이너 B의 후킹 모듈(110)로 기존 작업의 재개를 알린다. 기존 작업을 재개할 때는 후킹 모듈(110)은 `cudaMalloc()` 함수를 호출하여 다시 GPU 메모리를 할당받아 사용한다(240번 코드).
- [0072] 그리고 `cudaMemcpy()` 함수를 호출하여, 메인 메모리에 백업해 둔 데이터를 다시 GPU 메모리로 스왑-인(swap in)한다(250번 코드). 그리고 나서 메인 메모리에 임시로 할당해 둔 메모리의 할당을 해제한다(260번 코드). 이 과정이 끝나면 컨테이너 B는 GPU 메모리에서 이전에 실행하던 작업을 이어서 시작할 수 있다(150번 코드 이후).
- [0073] 여기서, 130번 코드 실행 직후에 B를 선점하여 스왑-아웃 하는 경우와 같이, 컨테이너에 일단 할당만 되고 한 번도 사용된 적이 없는 GPU 메모리를 스왑-아웃 하는 경우, GPU 메모리에 있는 데이터 값은 의미가 없다. 이 경우, 메인 메모리를 할당 받고 복사하는 과정(210, 220번 코드)을 생략하고, 바로 `cudaFree()`를 호출할 수 있다(230번 코드). 데이터의 복사 과정 없이 GPU 메모리를 선점한 경우에는, 스왑-인 할 때도 메모리 복사(250, 260번 코드)는 생략되고 `cudaMalloc()` 호출로 메모리를 할당 받는 것(240번 코드)만으로 스왑-인 할 수 있다.
- [0074] 한편, 메모리 변환 테이블(120)은 사용자 메모리 주소(`ptrUser`), 실제 메모리 주소(`ptrReal`), 메모리 할당 크기(`size`), 메모리 상태 플래그(`status`)를 저장한다.
- [0075] 사용자 메모리 주소(`ptrUser`)는 각 메모리 할당 오브젝트별로 사용자 어플리케이션에서 사용하는 주소로, 사용자 어플리케이션에서 처음 `cudaMalloc` 함수를 호출할 때, 후킹 모듈(110)이 설정하고 사용자 어플리케이션에 반환한 GPU 메모리 주소이다. 사용자 메모리 주소(`ptrUser`)는 각 메모리 할당 오브젝트 별로 사용자 어플리케이션에서 사용하는 메모리 주소로, 사용자가 메모리를 해제할 때까지 변하지 않고 유지된다.
- [0076] 실제 메모리 주소(`ptrReal`)는 현재 GPU 메모리 상에서 데이터가 처리되고 있는 경우, GPU 메모리 할당을 위해 최종적으로 CUDA API의 `cudaMalloc`을 호출했을 때 반환되는 실제 GPU 메모리 주소이다. 처음 반환된 GPU 메모리 주소와 최종적으로 반환된 GPU 메모리 주소는 상이할 수 있다. 그리고 스왑-아웃되어 데이터가 메인 메모리에 임시 저장되어 있다면, 실제 메모리 주소(`ptrReal`)는 임시 저장된 메인 메모리 주소가 된다.
- [0077] 메모리 할당 크기는 사용자 어플리케이션이 `cudaMalloc` 함수 호출 시 인자로 들어온 메모리 크기 값에 해당한다.
- [0078] 메모리 상태 플래그는 두 비트로 구성된다. 첫 번째 비트는 데이터가 현재 GPU 메모리 상에 있는지 메인 메모리 상에 있는지를 나타내는 비트이고, 두 번째 비트는 할당 받은 메모리가 사용된 적이 있는지 여부를 나타내는 비트이다.
- [0079] 즉, 첫 번째 비트는 현재 데이터가 GPU 메모리에서 처리되고 있으면 0, 메인 메모리에 임시로 저장되어 있으면 1로 표시된다. 그리고, 두 번째 비트는 메모리를 한 번도 사용한 적이 없으면 0, 한 번이라도 사용한 적이 있으면 1로 표시된다.
- [0080] 메모리 변환 테이블(120)은 사용자 어플리케이션에서 사용자 메모리 주소로 함수가 호출되더라도, 메모리 변환 테이블(120)에 사용자 메모리 주소와 일대 일 매핑되어 있는 실제 메모리 주소로 변환한다. 그리고 변환된 메모리 주소로 함수가 호출되도록 한다. 이에 대해서는 이후 상세히 설명한다.
- [0081] 통신 모듈(130)은 원격 프로시저 호출(RPC: Remote Procedure Call)을 통해 노드 스케줄러(200)의 통신 모듈(210)과 통신을 수행한다. 후킹부(100)는 노드 스케줄러(200)로 통신을 통해 메모리 할당 상태 보고, 메모리 해제 상태 보고, 메모리 신규 할당 요청 신호 등을 보낸다.
- [0082] 노드 스케줄러(200)는 현재 노드에서 실행 중인 적어도 하나 이상의 컨테이너들의 목록을 관리한다. 컨테이너 목록에는 컨테이너 식별자(`containerid`), 컨테이너가 등록된 시각(`starttime`), 해당 컨테이너와 통신한 마지막 통신 시각(`lasttime`), 컨테이너가 요청한 메모리 요구량(`memreq`), 컨테이너가 사용중인 메모리 사용량(`memusage`), 중단 여부 등 상태(`status`), 우선순위(`priority`) 등의 정보를 관리한다.
- [0083] 이러한 노드 스케줄러(200)는 통신 모듈(210), 컨테이너 식별자 생성 모듈(220), 컨테이너 리스트 테이블(230), 그리고 스케줄러(240)를 포함한다.
- [0084] 통신 모듈(210)은 원격 프로시저 호출(RPC: Remote Procedure Call)을 통해 후킹부(100)와 통신을 수행한다. 통신 모듈(210)은 후킹부(100)로부터 컨테이너 등록 신호, 메모리 할당 상태 보고, 메모리 해제 상태 보고, 메

모리 신규 할당 요청 신호 등을 수신한다. 그리고 통신 모듈(210)은 컨테이너 식별 정보, GPU 컨테이너 정지 신호, GPU 메모리 선점 명령 신호, 또는 메모리 신규 할당 허가 신호 등을 전송한다.

- [0085] 컨테이너 식별자 생성 모듈(220)은 후킹부(100)로부터 전송된 컨테이너 등록 요청 과정에서, 컨테이너에 할당할 컨테이너 식별자를 생성한다. 컨테이너 등록 요청 과정은 전처리 과정의 일부로, GPU 컨테이너가 새로 생성되고 사용자 프로그램에서 후킹 대상 CUDA API가 최초로 호출되면, 새로 생성된 GPU 컨테이너 자신을 노드 스케줄러(200)에 등록하는 과정을 의미한다. 컨테이너 식별자 생성 모듈(220)이 컨테이너 식별자를 생성하는 방법이나 컨테이너 식별자의 형태는 다양하므로, 본 발명의 실시예에서는 어느 하나의 방법이나 형태로 한정하지 않는다.
- [0086] 컨테이너 식별자 생성 모듈(220)은 생성한 컨테이너 식별자를 통신 모듈(210)을 통해 후킹부(100)로 전송한다. 후킹부(100)는 컨테이너 식별자를 수신한 후, 노드 스케줄러(200)로 메모리 할당 승인, 메모리 선점 승인, 또는 메모리 해제 알림 등의 신호를 전송할 때, 컨테이너 식별자를 포함하여 전송한다.
- [0087] 스케줄러(230)는 GPU 메모리에서 처리 가능한 작업들에 대한 우선순위 정보를 저장한다. 스케줄러(230)는 복수의 컨테이너들이 한정된 GPU 메모리를 공유하여 서로 사용하고자 할 때, 우선순위에 따라 이미 할당 받은 GPU 메모리를 해제시킬 컨테이너를 선택할 수 있다.
- [0088] 또한, 스케줄러(230)는 복수의 컨테이너들이 GPU 메모리를 각각 할당 받아 데이터를 처리할 수 있도록 하거나, 가장 높은 우선순위를 가지는 하나의 컨테이너만 GPU 메모리를 할당 받아 작업을 처리할 수 있도록 결정할 수도 있다.
- [0089] 즉, 스케줄러(230)는 다양한 스케줄링을 적용할 수 있도록 설계되어 있다. 기본적으로 현재 시스템 관리자의 목적이나 적용 시점의 상황에 따라 여러 컨테이너를 패킹하거나, 최우선 순위의 컨테이너 하나만 실행하고 나머지는 모두 일시 중단하는 두 가지의 스케줄링 정책을 제공할 수 있다. 따라서, 스케줄러(230) 내에 패킹 스케줄링 정책과 독점 스케줄링 정책 정보가 저장되어 있다.
- [0090] 상기에서 언급한 스케줄링 정책 이외에도 다양한 스케줄링 정책의 적용이 가능하다. 시스템 관리자는 목적에 맞는 커스텀 스케줄링 정책을 개발하여 GPU 메모리 스케줄러(10)에 쉽게 적용할 수 있다.
- [0091] 스케줄러(230)가 우선순위에 따라 GPU 메모리를 해제시킬 컨테이너를 선택하는 방법, 하나 또는 복수의 컨테이너들이 GPU 메모리를 할당 받도록 처리하는 방법 등은 다양한 기준으로 구현될 수 있으므로, 본 발명의 실시예에서는 어느 하나로 한정하지 않는다.
- [0092] 컨테이너 리스트 테이블(240)은 노드 스케줄러(200)에 연결되어 있는 복수의 컨테이너들 각각의 컨테이너 식별자, 컨테이너에서 요청한 메모리 요구량, 컨테이너에서 사용중인 메모리 사용량, 상태 정보를 저장, 관리한다.
- [0093] 후킹부(100)로부터 컨테이너 등록을 위한 등록 원격 프로시저 호출이 들어오면, 컨테이너 식별자 생성 모듈(220)은 해당 컨테이너의 식별자(containerid)를 생성하고, 이를 컨테이너 리스트 테이블에 등록한다. 이후 발생하는 모든 원격 프로시저 호출에 대해, 컨테이너 식별자를 사용하여 어떤 컨테이너가 원격 프로시저 호출을 하였는지 구별한다.
- [0094] 또한, 컨테이너 리스트 테이블의 등록 시점의 타임스탬프 값(starttime)을 저장한다. 요구량(Req, 혹은 필요량) 값은 해당 컨테이너의 사용자 어플리케이션이 GPU 메모리 할당을 시도한 총량이다. 여기에는 GPU 메모리 실제 사용량과 스왑-아웃된 메모리 크기를 모두 포함한다. 사용량(Usage) 값은 현재 실제로 GPU 메모리를 할당 받아 사용하고 있는 값이다. 여기에는 스왑-아웃된 메모리의 크기가 포함되지 않는다.
- [0095] 상태(status)값이 0이면 해당 컨테이너는 계속 실행되며, -1이면 해당 컨테이너는 후킹 모듈의 precall 전처리 루틴에서 더 이상 실행하지 않고 멈춘 상태를 유지하고 있음을 의미한다.
- [0096] 이상에서 설명한 GPU 메모리 스케줄러(10)를 이용하여 GPU 메모리를 선점하거나 스케줄링하는 방법에 대해 도 4 및 도 5를 참조로 설명한다.
- [0097] 도 4는 본 발명의 실시예에 따른 후킹부의 초기 절차에 대한 흐름도이다.
- [0098] 도 4에 도시된 바와 같이, 최초 GPU 컨테이너가 생성되면 후킹부(100)도 비활성 상태로 컨테이너 내부에 삽입되어 함께 생성된다. 그리고 해당 컨테이너의 후킹부(100)가 사용자 어플리케이션에 의한 CUDA API 호출이 발생함을 인지하면(S100), 발생한 CUDA API 호출이 최초 호출인지 확인한다(S110). CUDA API 호출이 최초 호출인지 판단하는 방법은 다양하므로, 본 발명의 실시예에서는 상세한 설명을 생략한다.

- [0099] CUDA API 호출이 최초 호출인 경우, 후킹부(100)는 비활성 상태에서 활성 상태로 상태를 천이하고(S120), 노드 스케줄러(200)로 후킹부(100)가 포함되어 있는 해당 컨테이너의 등록을 요청한다(S130). 그리고 노드 스케줄러(200)로부터 컨테이너 식별자를 수신하면(S140), 이를 저장, 관리한다.
- [0100] S110 단계에서 확인한 결과, S100 단계에서 인지한 API 호출이 최초 CUDA API 호출이 아닌 것으로 확인하면, 기 저장한 컨테이너 식별자를 토대로 GPU 메모리를 선점하거나 새로 할당 받아 작업을 처리하는 절차를 수행한다.
- [0101] 다음은 GPU 메모리를 선점하여 사용하는 방법에 대해 도 5를 참조로 설명한다. 도 5를 설명하기 앞서, 본 발명의 실시예에서는 제1 컨테이너에 포함되어 있는 제1 후킹부가 미리 GPU 메모리를 할당 받아 작업을 처리하고 있는 상태에서, 제2 컨테이너에 포함되어 있는 제2 후킹부가 GPU 메모리를 선점하는 방법을 예로 하여 설명한다.
- [0102] 도 5는 본 발명의 실시예에 따른 GPU 메모리 선점 방법에 대한 흐름도이다.
- [0103] 도 5에 도시된 바와 같이, 제1 컨테이너가 이미 GPU 메모리를 할당 받아 사용자 어플리케이션에서 호출한 함수를 토대로 작업을 처리하고 있다(S200). 이때, 제2 컨테이너의 사용자 어플리케이션이 GPU 메모리를 할당 받기 위해 cudaMalloc 함수를 발생하여 CUDA API로 전송하면, 제2 후킹부(100-2)는 CUDA API로 전달되는 API를 확인한 후 cudaMalloc 호출을 가로챈다(S201).
- [0104] 이와 동시에 제2 후킹부(100-2)는 메모리 변환 테이블을 생성한다. 이때, cudaMalloc에는 사용자 어플리케이션이 사용하고자 하는 메모리 용량, 어떤 작업을 수행할 것인지에 대한 작업 정보, 사용자 메모리 주소 등이 변수로 포함되어 있다.
- [0105] 제2 후킹부(100-2)는 노드 스케줄러(200)로부터 할당 받은 컨테이너 식별자를 포함하여 노드 스케줄러로 메모리 할당이 가능한지 질의한다(S202). 노드 스케줄러(200)는 컨테이너 리스트 테이블을 참조로 S202 단계에서 제2 후킹부(100-2)가 질의한 메모리 할당 질의에 포함된 메모리 용량만큼 제2 컨테이너에 할당할 수 있는지 확인한다(S203).
- [0106] GPU 메모리 용량의 대부분을 제1 후킹부(100-1)에서 사용하고 있고, 제2 후킹부(100-2)가 질의한 사용자 어플리케이션의 작업이 제1 컨테이너에서 현재 처리하고 작업보다 우선순위가 높다고 가정한다. 그러면, 노드 스케줄러(200)는 제1 후킹부(100-1)로 GPU 메모리 사용을 중지하는 실행 정지 명령을 전송한다(S204).
- [0107] 제1 후킹부(100-1)는 노드 스케줄러(200)로부터 수신한 실행 정지 명령에 따라 GPU 메모리에서 처리하던 작업을 정지한다. 그리고, GPU 메모리에서 처리된 데이터를 메인 메모리로 스왑-아웃하기 위해, 메인 메모리 할당을 위한 API를 호출한다(S205). 메인 메모리가 할당되면, 제1 후킹부(100-1)는 GPU 메모리에 있는 데이터를 메인 메모리로 스왑 아웃하고, cudaFree 함수를 호출한다(S206).
- [0108] 제1 후킹부(100-1)는 노드 스케줄러(200)로 제1 컨테이너의 사용자 어플리케이션이 이용하던 GPU 메모리가 반환되었음을 알리고(S207), 노드 스케줄러(200)로부터 또 다른 명령이 발생할 때까지 대기한다(S208).
- [0109] 노드 스케줄러(200)는 S207 단계에서 GPU 메모리 반환 알림 신호를 수신하면, 제2 후킹부(100-2)로 메모리 할당 허가 신호를 전송한다(S209). 그리고 노드 스케줄러(200)는 컨테이너 리스트 테이블을 갱신한다(S210).
- [0110] 제2 후킹부(100-2)는 사용자 어플리케이션에서 수신한 cudaMalloc API를 CUDA API로 전달하여 GPU 메모리를 할당 받도록 한다(S211). GPU 메모리가 할당되면 제2 컨테이너의 사용자 어플리케이션은 GPU 메모리를 이용하여 작업을 처리한다(S212). 본 발명의 실시예에서는 설명의 편의를 위하여 작업 처리를 제2 후킹부(100-2)에서 실행하는 것으로 표현한다.
- [0111] 제2 컨테이너의 사용자 어플리케이션이 GPU 메모리를 이용한 작업 처리를 완료한 후 cudaFree를 호출한다. 제2 후킹부(100-2)는 사용자 어플리케이션이 호출한 cudaFree를 가로채고, 노드 스케줄러(200)로 GPU 메모리가 해제되었음을 알린다(S213, S214).
- [0112] 노드 스케줄러(200)는 S214 단계에서 수신한 GPU 메모리 해제 알림 신호를 확인한 후, 컨테이너 리스트 테이블에 등록되어 있는 정보를 확인한다. 그리고 이 전에 작업이 중지된 제1 컨테이너의 제1 후킹부(100-1)로 실행 재개 명령을 전달한다(S215).
- [0113] 제1 후킹부(100-1)는 중지된 작업을 재개하기 위하여 GPU 메모리를 할당을 위한 cudaMalloc 함수를 호출하여 CUDA API로 전달한다. GPU 메모리가 할당되면 제1 후킹부(100-1)는 메인 메모리에 임시 저장한 데이터를 GPU 메모리로 복사하는 스왑 인을 위해 cudaMemcpy 함수를 호출한다. 메인 메모리에 임시 저장된 데이터는 GPU 메모리

로 스왑-인 되고, GPU 메모리를 이용하여 임시 저장되었던 데이터 이후의 데이터부터 작업을 처리한다(S217).

- [0114] 이때, S216 단계에서 호출된 `cudaMalloc` 함수에 의해 GPU 메모리에서 할당 받은 메모리 주소가 S200 단계에서 할당 받은 메모리 주소와 다를 수 있다. 따라서, 제1 후킹부(100-1)의 메모리 변환 테이블(120)은 사용자 어플리케이션에서 전송하는 사용자 메모리 주소를 새로 할당 받은 실제 메모리 주소로 변환한 후 CUDA API로 전달해야 한다. 이에 대해 도 6을 참조로 설명한다.
- [0115] 도 6은 본 발명의 실시예에 따른 사용자 메모리 주소 충돌을 나타낸 예시도이다.
- [0116] 상기 도 3의 후킹부(100)에서 메모리 변환 테이블(120)에 표시한 것과 같이, 사용자 어플리케이션은 사용자 메모리 주소(`ptrUser`)의 주소 공간을 사용하며, CUDA API는 실제 메모리 주소(`ptrReal`)의 주소 공간을 사용한다. 따라서, 사용자 어플리케이션이 사용하는 모든 CUDA API 호출에 대해 인자로 주소가 들어오면, 메모리 변환 테이블(120)은 이를 치환해 주어야 한다.
- [0117] 이를 위해 `cudaMalloc` 호출 시 메모리 변환 테이블(120)에 메모리 관련 정보를 등록하여 관리한다. 메모리 복사 함수인 `cudaMemcpy`가 호출되면 복사 방향이 GPU 메모리-메인 메모리인지, 메인 메모리-GPU 메모리인지를 확인한다. 메모리 변환 테이블(120)은 GPU 메모리 주소인 인자를 찾고, 이를 기반으로 해당 주소를 사용자 메모리 주소(`ptrUser`)에서 찾아 실제 메모리 주소(`ptrReal`)로 치환한다.
- [0118] 후킹 모듈이 실제 CUDA API를 호출할 때는 실제 메모리 주소(`ptrReal`) 값을 인자로 넣어 호출한다. `cudaFree`의 경우에도 마찬가지로 인자로 들어온 GPU 메모리 주소를 메모리 변환 테이블(120)의 정보를 사용하여 치환해 CUDA API를 호출한다.
- [0119] 커널 실행의 경우, `cudaSetupArgument` 호출을 사용하여 인자들을 호출 스택에 넣고, `cudaLaunch`로 커널을 실행하는 방식을 사용한다. 따라서 `cudaSetupArgument` 호출을 가로채, 인자 크기가 메모리 주소와 동일하지 확인한다. 만약 인자 크기가 메모리 주소와 동일하다면 해당 인자가 메모리 변환 테이블(120)의 사용자 메모리 주소(`ptrUser`) 범위에 존재하는지 확인한다. 만약 존재한다면, 이를 실제 메모리 주소의 값과 사용자 메모리 주소의 값의 차이(`ptrReal-ptrUser`)만큼, 사용자 메모리 주소에 더해 사용자 메모리 주소를 실제 메모리 주소로 변환하여 CUDA API의 `cudaSetupArgument`를 호출한다.
- [0120] 그러나, 사용자 어플리케이션에서 내려온 주소가 GPU 메모리 주소가 아니라면, 후킹 모듈(110)은 사용자 어플리케이션으로부터 내려온 인자 값 그대로 CUDA API의 `cudaSetupArgument`를 호출한다.
- [0121] 상기 도 3에서 후킹부(100)의 메모리 변환 테이블(120)에 화살표로 표시한 경우를 예로 하여 설명하면 다음과 같다. 사용자 어플리케이션이 0x1000번지 주소의 GPU 메모리로 접근을 요청했다면, 메모리 변환 테이블(120)은 0x1000번지의 사용자 메모리 주소(`ptrUser`) 값을 갖는 메모리 오브젝트를 찾는다. 그리고 이를 그에 대응하는 실제 메모리 주소(`ptrReal`) 값인 0x2000번지로 치환해 준다. 그 후 실제 CUDA API 호출 시에는 0x1000 대신 0x2000을 인자로 사용하게 된다.
- [0122] 이는 사용자 어플리케이션이 접근을 요청한 주소가 `ptrUser` 값과 반드시 동일해야 한다는 의미가 아니며, `ptrUser` 내지 (`ptrUser+size`)의 범위 내에 있다면 문제 없다. 사용자 어플리케이션이 접근을 요청한 주소가 이 범위 내에 있고, `ptrUser`와 `offset` 만큼의 차이가 있어 (`ptrUser+offset`)으로 표현 가능하다면, 해당 주소는 (`ptrReal+offset`)으로 치환한다.
- [0123] 이때, 사용자 메모리 주소(`ptrUser`) 값의 경우, 최초 할당된 실제 메모리 주소(`ptrReal`) 값을 그대로 사용할 경우 주소 공간이 충돌할 수 있다.
- [0124] 즉, 도 6에 도시된 바와 같이, 최초 할당된 A의 주소가 10이고, B의 주소가 15라 가정한다. B가 스왑-아웃 된 상태에서 A가 할당 해제되고, B가 그 이후에 스왑-인 되면, A가 사용하던 10번지는 할당 해제된 상태이므로 B가 10번지에 스왑-인 될 수 있다. 그렇다면 B는 사용자 메모리 주소(`ptrUser`)가 15이고, 실제 메모리 주소(`ptrReal`)는 10이다.
- [0125] 이 상황에서 새로운 메모리 C가 할당된다면 실제 주소 기준으로 15는 비어있는 상태이므로, C가 15에 할당될 수 있다. 만약 최초 할당된 실제 메모리 주소(`ptrReal`) 값을 사용자 메모리 주소(`ptrUser`)에 그대로 사용한다면, C의 사용자 메모리 주소(`ptrUser`)는 15가 되어야 한다.
- [0126] 그런데, B의 사용자 메모리 주소(`ptrUser`)가 15이므로, B와 C의 사용자 메모리 주소(`ptrUser`)가 서로 충돌해 문제가 발생한다. 이러한 문제를 해결하기 위해, 사용자 메모리 주소(`ptrUser`)는 실제 메모리 주소(`ptrReal`)와는

별개의 주소 공간으로서, 실제 메모리 주소(ptrReal)와는 연관이 없는 값을 사용한다.

- [0127] 따라서 사용자 어플리케이션에서 사용하는 모든 GPU 주소는 메모리 변환 테이블(120)의 정보를 통해 실제 주소로 치환되어야 한다. 이는 스왑-아웃된 적이 없는 메모리 오브젝트의 경우에도 동일하게 적용되는 것을 예로 한다.
- [0128] 도 7은 본 발명의 실시예에 따른 오브젝트 단위 GPU 메모리 스왑에 대한 예시도이다.
- [0129] 사용자 어플리케이션이 GPU 메모리를 선점할 때, 항상 선점 대상 컨테이너가 할당 받은 GPU 메모리 전체를 스왑하는 것은 비효율적이다.
- [0130] 예를 들어, GPU 메모리를 선점할 때 필요한 메모리의 크기가 충분히 작아서, 선점 대상 컨테이너가 사용 중인 GPU 메모리 전체를 할당 해제하지 않아도 그 목적을 달성할 수 있는 경우가 있다. 이러한 경우, 필요한 만큼만 스왑해야, 불필요한 스왑-아웃과 스왑-인으로 인한 메모리 복사 오버헤드를 줄일 수 있다.
- [0131] GPU 메모리 스케줄러(10)는 후킹 모듈(110)의 메모리 변환 테이블(120)에서 GPU 메모리 할당을 오브젝트(Object) 단위로 전부 관리하므로, 컨테이너 단위가 아닌 오브젝트 단위로 메모리를 스왑하여 오버헤드를 줄일 수 있다.
- [0132] 도 7에 도시된 바와 같이, A부터 G까지는 컨테이너들이 할당을 시도한 GPU 메모리이다. 컨테이너 단위로 메모리를 스왑하는 경우, 컨테이너 3처럼 모든 메모리가 GPU 메모리에 있거나, 컨테이너 2처럼 모든 메모리가 스왑-아웃되어 메인 메모리에 있게 된다.
- [0133] 즉, 컨테이너 단위로 메모리를 스왑한다면, A가 할당될 GPU 메모리 공간이 있음에도 불구하고 컨테이너 1의 메모리가 모두 스왑-아웃되기 때문에 A도 함께 스왑-아웃된다. 그리고 이후 작업 재개시에도 A를 다시 스왑-인 해야 하므로, 불필요한 메모리 복사가 발생한다.
- [0134] 그러나 메모리를 오브젝트 단위로 스왑할 경우, 컨테이너 1과 같이 일부는 GPU 메모리에 할당되고, 일부는 메인 메모리에 할당될 수 있다. 즉, 오브젝트 단위 스왑을 적용하면 A와 같은 경우에 스왑이 발생하지 않으므로, 불필요한 메모리 복사를 막을 수 있다.
- [0135] GPU 커널의 실행 시간은 보통 마이크로초(μs)나 밀리초(ms) 단위이며, 길어야 2초 내지 3초 이내이다. GPU 커널 실행 도중에는 레지스터나 공유 메모리 등의 수명이 끝나지 않아 아직 유효한 상태를 유지한다. 따라서, 유효한 상태를 유지하고 있는 GPU 커널을 강제로 중단하려는 경우, GPU 내부 컨텍스트의 정보도 함께 백업해야 한다. 이는 한 커널이 끝날 때까지의 짧은 시간을 대기하는 것보다 큰 이득이 있다고 보기 어렵다.
- [0136] 또한, NVIDIA GPU는 커널 실행을 임의로 중단하는 방법이 존재하지 않으며, 내부의 캐시나 레지스터 등의 데이터를 백업하는 방법도 없다. 단일 커널의 실행 시간이 충분히 짧은 것과, 이를 중단했을 때 발생하는 오버헤드 등을 종합적으로 고려할 때, GPU 메모리의 정지나 선점을 위해서는 커널 단위로 진행하는 것이 좋다. 따라서, 본 발명의 실시예에서는 커널이 실행되고 있지 않을 때, 후킹부(100)에서 GPU 메모리를 정지하거나 선점할 수 있도록 한다.
- [0137] 다음은 본 발명의 스케줄러(240)에서 제공하는 패킹 스케줄링 정책에 대해 도 8을 참조로 설명한다.
- [0138] 도 8은 본 발명의 실시예에 따른 패킹 스케줄링 정책의 예시도이다.
- [0139] 도 8에 도시된 바와 같이, 스케줄러(240)는 먼저 전체 GPU 컨테이너들을 미리 설정한 기준에 따라 정렬한다. 여기서 사용되는 기준은 사용률 최대화나 평균 대기시간 최소화 등 시스템 관리자의 목적에 따라, 컨테이너의 생성 시각, 우선순위, 메모리 사용량, 메모리 요구량, 메모리 오브젝트 수 등 어떤 요소라도 기준이 될 수 있다. 그리고, 단순히 오름차순, 내림차순뿐만 아니라 필요에 따라 여러 조건을 복합적으로 적용하는 것 또한 가능하다.
- [0140] 이렇게 정렬된 컨테이너 목록 $C=\{C_1, C_2, \dots, C_N\}$ 은 순서대로 메모리 할당 서열이 된다. 그 후, 메모리 할당 서열이 높은 컨테이너인 C_1 부터 순서대로 메모리 요구량을 더한다. 여기서 메모리 요구량이란 해당 컨테이너가 필요로 하는 총 GPU 메모리 크기를 말하며, 스왑-아웃 되었거나, 아직 할당되지 못한 신규 메모리 할당 요청의 크기를 모두 포함한다(①).
- [0141] 이때, 임의의 컨테이너(C_{k+1})의 메모리 요구량을 허용하기에는 GPU 메모리가 부족한 상황이 발생할 수 있다(②). 그러면 스케줄러(240)는 물리적 GPU 메모리 크기를 초과하지 않는 범위 내에서 최대한 더한 것이 몇 번째 컨테

이너까지인지 확인한다. 확인된 컨테이너가 C_k 컨테이너까지라고 하면, C_k 컨테이너까지의 컨테이너의 메모리 할당은 모두 유지되며, 추가 할당 요청도 모두 승인된다.

[0142] 그 후에는, 다시 C_{k+1} 부터 순서대로 현재 메모리 사용량만을 더한다(③). 잔여 메모리 크기 범위 내에서 최대 C_L 컨테이너까지의 현재 메모리 사용량을 감당할 수 있다면, C_{k+1} 부터 C_L 컨테이너까지는 기존 메모리를 선점당하지 않고, 현재 메모리 할당 상태는 유지된다. 그러나, 추가 메모리 할당은 거부되며, 실행도 멈춘다.

[0143] C_{L+1} 컨테이너의 메모리는 최종적으로 남은 잔여 메모리 크기만큼만 GPU 메모리 점유가 허가되며, 이를 뺀 나머지는 스왑-아웃된다(④). C_{L+2} 컨테이너부터 C_N 컨테이너까지는 기할당된 모든 메모리가 스왑-아웃 된다(⑤).

[0144] 이상에서 설명한 GPU 메모리 스케줄러(10)가 GPU의 메모리를 선점하여 스케줄링하는 환경에서, 실제 GPU 메모리 스케줄러(10)가 GPU의 메모리를 선점함으로써 GPU 메모리 사용량 변화와 GPU 사용률 변화에 대해 도 9를 참조로 설명한다.

[0145] 도 9는 본 발명의 실시예에 따라 시간에 따른 GPU 메모리 사용량 및 GPU 사용률 변화를 나타낸 그래프이다.

[0146] GPU 메모리 사용량 변화와 사용률 변화를 실험한 실험 환경은 다음 표 3과 같다.

표 3

[0147]	메인 메모리	64GB
	GPU	NVIDIA Quadro P4000
	GPU 메모리	8GB
	CUDA	NVIDIA CUDA 9.0 (Patch 4)
	Docker	Docker version 18.09.6, build 481bc77
	nvidia-docker	NVIDIA Docker: 2.0.3

[0148] 본 발명의 실시예에서는 노드 스케줄러(200)가 컨테이너 생성 시각의 내림차순으로 메모리 할당 서열을 부여하여, 나중에 생성된 컨테이너의 우선순위가 더 높게 하였다. 이 실험에는 약 2초가 소요되는 행렬 곱 커널을 사용하였으며, GPU 메모리는 1,500MB 3개씩을 할당 받도록 하였다. 행렬 곱 커널을 4회 실행하는 컨테이너 1을 실행하고, 약 3초 후, 행렬 곱 커널을 1회 실행하는 컨테이너 2를 실행하였다.

[0149] 도 9의 (a)는 GPU 메모리 사용량의 변화를 나타낸 그래프이고, 도 9의 (b)는 GPU 사용률 변화를 나타낸 그래프이다.

[0150] 도 9에 도시된 바와 같이, 컨테이너 1이 먼저 GPU를 사용하기 시작하였고, 약 3초 후부터 컨테이너 2가 실행되었다. 컨테이너 1과 컨테이너 2는 모두 각각 4,583MB의 GPU 메모리를 필요로 한다.

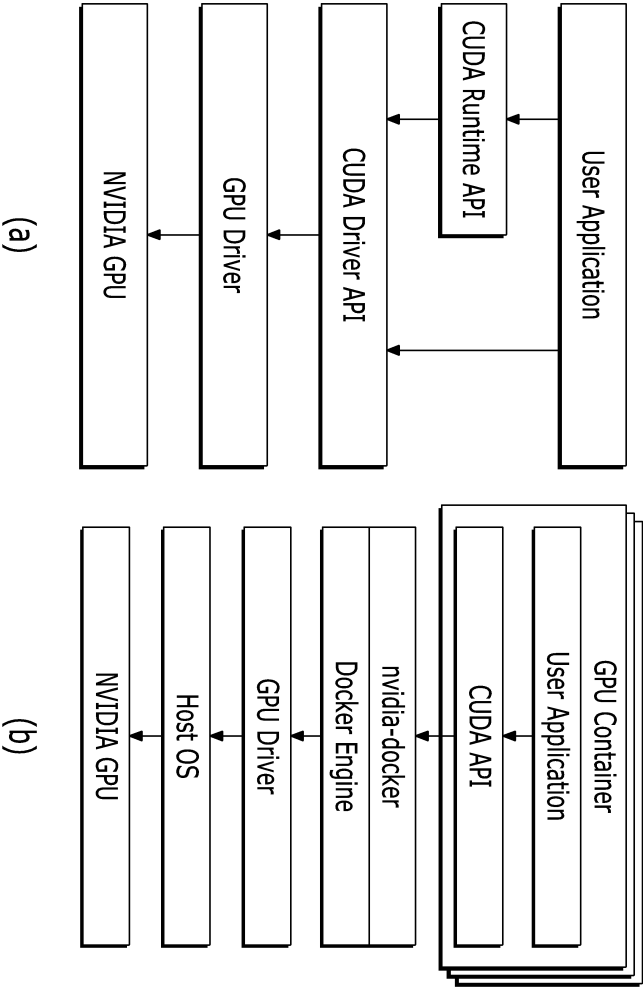
[0151] CUDA 프로그램 실행 시 GPU에 필요한 10MB를 포함하여 총 9,176MB의 메모리가 필요하지만, 실험 환경 상 GPU의 메모리 크기는 8,106MB이므로 모두 할당하기에는 GPU 메모리가 부족하다. 따라서 컨테이너 1의 GPU 사용은 4초 경부터 중단되었고, 컨테이너 1의 GPU 메모리는 4-5초 경에 스왑-아웃 되었으며, 컨테이너 2는 5초경에 필요한 GPU 메모리를 모두 선점하여 할당 받았음을 알 수 있다.

[0152] 또 컨테이너 2는 5초경부터 약 2초간 GPU를 선점하여 사용하였다. 그 후, 컨테이너 2의 실행이 7초경에 끝나고, 7-8초 경에 컨테이너 1이 GPU 메모리를 다시 스왑-인 하였으며, 8-9초 경부터 약 6초간 다시 GPU를 사용하고 15초경에 종료되었다. 이를 통해 GPU 메모리 스케줄러가 GPU 연산과 GPU 메모리의 선점을 모두 포함하는 GPU 선점 기능을 제공함을 확인할 수 있다.

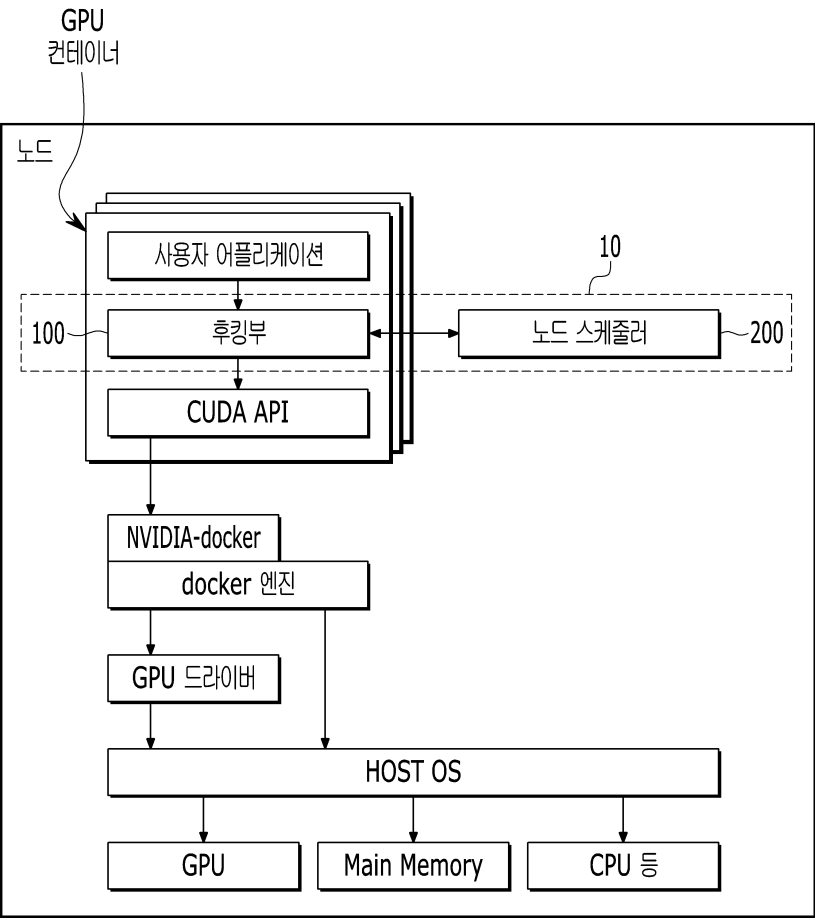
[0153] 이상에서 본 발명의 실시예에 대하여 상세하게 설명하였지만 본 발명의 권리범위는 이에 한정되는 것은 아니고 다음의 청구범위에서 정의하고 있는 본 발명의 기본 개념을 이용한 당업자의 여러 변형 및 개량 형태 또한 본 발명의 권리범위에 속하는 것이다.

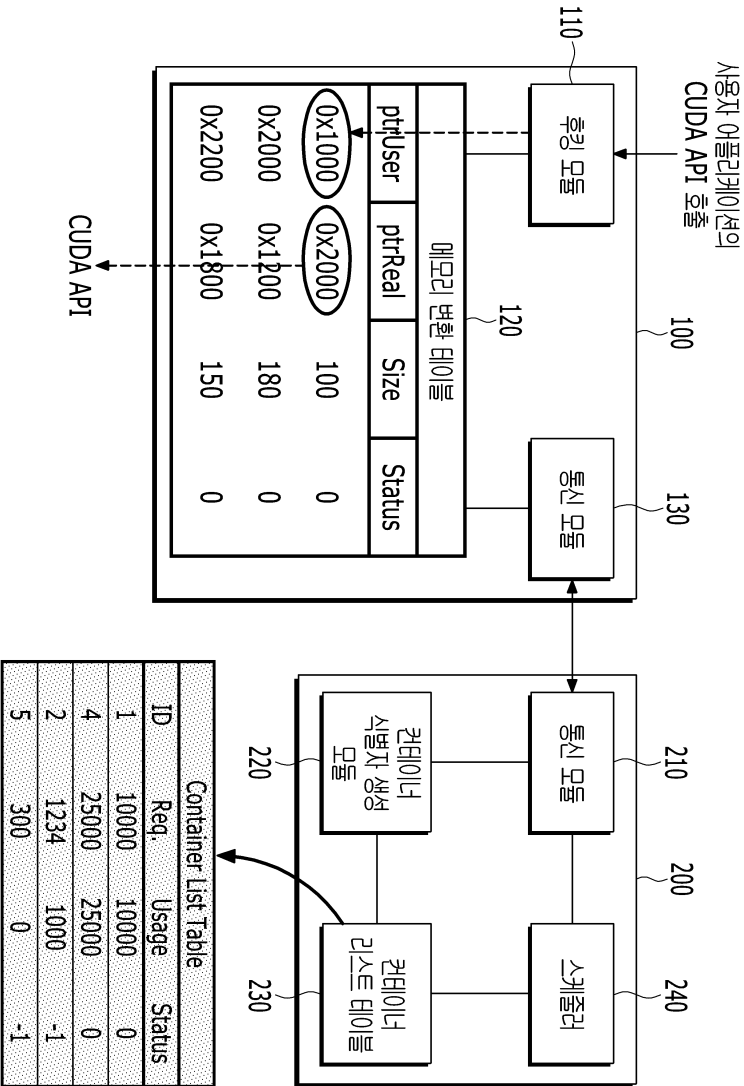
도면

도면1



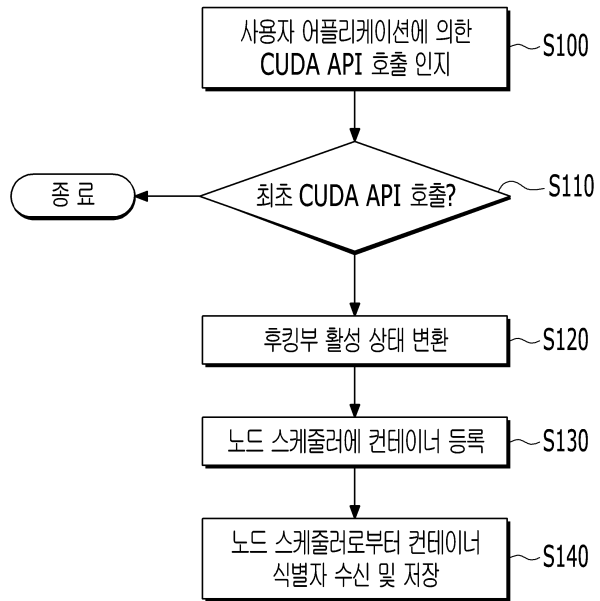
도면2



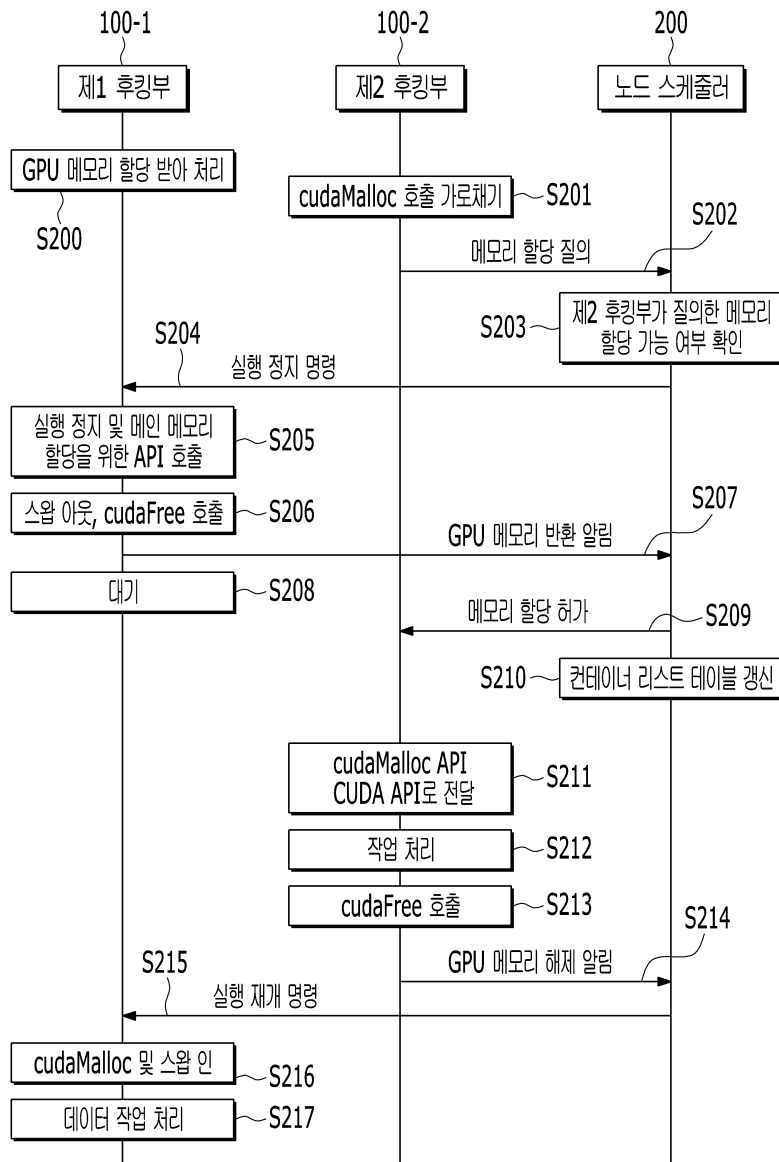


도면3

도면4



도면5



도면6

1. 최초 상태				
Memory	ptrUser	ptrReal	Status	
A	10	10	GPU memory	
B	15	15	GPU memory	

3. A의 할당 해제				
Memory	ptrUser	ptrReal	Status	
A			Free	
B	15	1234	Main memory	

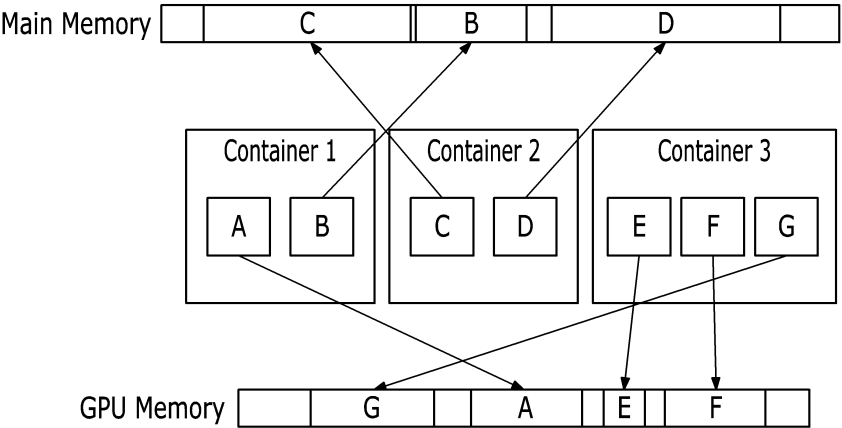
5-1. C의 신규 할당 (충돌 발생)				
Memory	ptrUser	ptrReal	Status	
A			Free	
B	15	10	GPU memory	
C	15	15	GPU memory	

2. B의 스왑-아웃				
Memory	ptrUser	ptrReal	Status	
A	10	10	GPU memory	
B	15	1234	Main memory	

4. B의 스왑-인				
Memory	ptrUser	ptrReal	Status	
A			Free	
B	15	10	GPU memory	

5-2. C의 신규 할당 (충돌 발생하지 않음)				
Memory	ptrUser	ptrReal	Status	
A			Free	
B	15	10	GPU memory	
C	30	15	GPU memory	

도면7



도면8

Container List			
Container	Requirement	Usage	Status
C_1	가능		실행
C_2	가능		실행
C_3	가능		실행
...	① ...	...	...
C_K-1	가능		실행
C_K	↓ 가능		실행
C_K+1	② 허용 시 메모리 초과	허용	중단, 메모리 유지
...	... ③	...	...
C_L-1		허용	중단, 메모리 유지
C_L		↓ 허용	중단, 메모리 유지
C_L+1	④ 일부 허용		중단, 메모리 일부 유지
C_L+2		0	중단, 메모리 회수
...	... ⑤	...	...
C_N		0	중단, 메모리 회수

도면9

