KIPS Trans. Comp. and Comm. Sys.
Vol.11, No.10 pp.333~344
ISSN: 2287-5891 (Print), ISSN 2734-049X (Online)

https://doi.org/10.3745/KTCCS.2022.11.10.333

GPU Resource Contention Management Technique for Simultaneous GPU Tasks in the Container Environments with Share the GPU

Jihun Kang†

ABSTRACT

In a container-based cloud environment, multiple containers can share a graphical processing unit (GPU), and GPU sharing can minimize idle time of GPU resources and improve resource utilization. However, in a cloud environment, GPUs, unlike CPU or memory, cannot logically multiplex computing resources to provide users with some of the resources in an isolated form. In addition, containers occupy GPU resources only when performing GPU operations, and resource usage is also unknown because the timing or size of each container's GPU operations is not known in advance. Containers unrestricted use of GPU resources at any given point in time makes managing resource contention very difficult owing to where multiple containers run GPU tasks simultaneously, and GPU tasks are handled in black box form inside the GPU. In this paper, we propose a container management technique to prevent performance degradation caused by resource competition when multiple containers execute GPU tasks simultaneously. Also, this paper demonstrates the efficiency of container management techniques that analyze and propose the problem of degradation due to resource competition when multiple containers execute GPU tasks simultaneously through experiments.

Keywords : HPC Cloud, Container, GPU Computing, GPU Sharing, Resource Race

GPU를 공유하는 컨테이너 환경에서 GPU 작업의 동시 실행을 위한 GPU 자원 경쟁 관리기법

강 지 훈†

요 약

컨테이너 기반 클라우드 환경은 다수의 컨테이너가 GPU(Graphic Processing Unit)를 공유할 수 있으며, GPU 공유는 GPU 자원의 유휴 시간을 최소화하고 자원 사용률을 향상할 수 있다. 하지만, GPU는 전통적으로 클라우드 환경에서 CPU, 메모리와는 다르게 컴퓨팅 자원을 논리적으로 다중화하고 사용자에게 자원 일부를 격리된 형태로 제공할 수 없다. 또한, 컨테이너는 GPU 작업을 실행할 때만 GPU 자원을 점유하며, 각 컨테이너의 GPU 작업 실행 시점이나 작업 규모를 미리 알 수 없기 때문에 자원 사용량 또한 미리 알 수 없다. 컨테이너가 GPU 자원을 임의의 시점에 제한 없이 사용한다는 특징은 다수의 컨테이너가 GPU 작업을 동시에 실행하는 환경에서 자원 경쟁 상태 관리를 매우 어렵게 만들며, GPU 작업은 대부분 GPU 내부에서 블랙박스 형태로 처리되기 때문에 GPU 작업이 실행된 이후에는 GPU 자원 경쟁을 방지하는데 제한적이다. 본 논문에서는 다수의 컨테이너가 GPU 작업을 동시에 실행할 때 자원 경쟁으로 인해 발생하는 성능 저하를 방지하기 위한 컨테이너 관리기법을 제안한다. 또한, 본 논문에서는 실험을 통해 다수의 컨테이너가 GPU 작업을 동시에 실행할 때 자원 경쟁으로 인한 성능 저하 문제를 분석하고 제안하는 컨테이너 관리기법의 효율성을 증명한다.

키워드 : 고성능 클라우드, 컨테이너, GPU 컴퓨팅, GPU 공유, 자원 경쟁

1. 서 론

GPU(Graphic Processing Unit)는 대규모 병렬 처리 기술을 활용하여 AI 학습 및 추론, 빅데이터 처리와 같은 고용량 작업의 고속 처리가 가능하다. GPU의 장점을 자원 확장성이 높은 클라우드 환경에서 활용하면, 고성능 작업을 실행할 때, 고성능 서버 인프라를 직접 구축하지 않아도 클라우드 서버에서 가용할 수 있는 GPU 자원의 용량 내에서 제한 없이 활용할 수 있다.

또한, 컨테이너 기반 클라우드 환경은 다수의 컨테이너가 단일 GPU를 공유할 수 있으며, 다수의 컨테이너에서 실행되는 GPU 작업을 다중 프로세스 방식으로 동시에 실행할 수 있기 때문에 GPU 자원의 활용률을 높일 수 있다. 이를

※ 이 논문은 2022년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임(2022R1I1A1A01063551).
† 정 회 원 : 고려대학교 4단계 BK21 컴퓨터학교육연구단 연구교수
Manuscript Received : May 25, 2022
Accepted : June 23, 2022
* Corresponding Author : Jihun Kang(k2j23h@korea.ac.kr)

통해 GPU 자원의 유휴 시간을 최소화하고 자원 사용률을 향상할 수 있다. 특히, 컨테이너나 가상머신과 같은 가상 인스턴스가 실행 중이기만 해도 일정량의 자원이 사용되는 CPU나 메모리와 같은 전통적인 컴퓨팅 장치와는 다르게 GPU의 경우 가상 인스턴스가 GPU 작업을 실행해야지만 사용되기 때문에 GPU를 한 명의 사용자에게만 할당한다면 자원이 유휴 상태가 될 확률이 높다. 반면 다수의 사용자가 GPU를 공유한다면 단일 GPU에서 다수의 GPU 작업을 동시에 실행하기 때문에 GPU 사용률이 증가하고 유휴 시간을 감소시킬 수 있다.

GPU 자원의 공유는 일반적으로 클라우드 환경에서 자원 공유로 인해 얻을 수 있는 장점들을 모두 얻을 수 있다. 특히, GPU의 경우 앞서 설명한 것과 같이 상대적으로 다른 컴퓨팅 장치보다 유휴 상태가 될 확률이 높기 때문에 자원 공유는 GPU 자원 활용률 향상을 위한 효과적인 방법이다. 또한, 클라우드 환경에서 GPU는 비용이 매우 높은 컴퓨팅 장치에 속한다. 비용이 많이 들지만 앞서 설명한 것과 같이 유휴 상태가 될 확률이 상대적으로 높아서 GPU의 활용률을 높이고 비용을 절감하기 위해서는 GPU 공유를 통해 여러 사용자의 GPU 작업을 함께 처리하는 것이 효과적이다. GPU 자원의 공유는 앞서 설명한 것과 같은 다양한 장점이 존재하지만, 다수의 사용자가 GPU를 공유할 때 전체 성능에 영향을 주는 몇 가지 고려해야 할 특징이 존재한다.

컨테이너 기반 클라우드 환경에서 GPU는 일반적으로 컨테이너에 GPU 자원의 일부만을 할당하기 위한 자원 격리나 컨테이너의 GPU 자원 사용량을 제한할 수 있는 기술이 제공되지 않는다[1]. 이로 인해 GPU를 공유하는 컨테이너 사이에서 GPU 자원을 균등하게 분할 할당할 수 없으며, 다수의 GPU 작업이 동시에 실행되면 자원 사용에 대한 제한 없이 각 GPU 작업은 프로그램 코드에 명시된 대로 GPU 자원이 필요한 만큼 모두 사용하도록 작동하기 때문에 컨테이너 사이에서 자원 경쟁이 발생한다. 또한, 다수의 사용자가 실행하는 GPU 작업으로 인한 자원 경쟁은 자원 초과 사용, 컨텍스트 전환의 증가, 입출력 작업 경쟁과 같은 다양한 요소로 인해 성능 저하가 발생하며 GPU를 공유하는 사용자가 증가할수록 자원 경쟁으로 인한 성능 영향은 더 커진다.

GPU 작업의 경우 GPU 내부에서 블랙박스 형태로 처리되기 때문에 클라우드 관리 시스템에서 자원 독점이나 경쟁에 대한 관리가 불가능하다. 특히, GPU를 공유하는 각 사용자는 GPU 작업을 독립적으로 운용한다. 따라서 각 사용자는 GPU 자원 사용을 위해 사용자 사이에 상호 어떠한 합의도 하지 않으며, GPU 작업을 실행하는 컨테이너는 알 수 없는 임의의 시간에 GPU 작업을 실행하기 때문에 자원 경쟁 상태를 관리하는데 제한적이다.

GPU 자원 공유는 GPU의 유휴 시간을 최소화하고 활용률을 높일 수 있다는 장점이 있지만, 자원을 공유하는 대상이 증가할수록 자원 경쟁 문제가 발생하며, 이는 컨테이너에서 처리하는 GPU 작업의 성능에 영향을 미친다. 자원 경쟁으로 인한 성능 영향을 방지하기 위해 클라우드 시스템 관리자는 GPU 자원 공유 기능을 활용할 때 자원 경쟁 상태는 완화해서 자원 경쟁으로 인한 성능 영향을 최소화해야 한다.

본 논문에서는 다수의 사용자가 GPU를 공유하는 컨테이너 기반 클라우드 환경에서 GPU 자원 경쟁으로 인한 성능 영향을 완화하기 위한 컨테이너 관리기법을 제안한다. 본 논문은 제안하는 기법을 설명하기 위해 실험을 통해 다수의 컨테이너에서 실행되는 GPU 작업이 단일 GPU에서 동시에 실행될 때 자원 경쟁으로 인한 성능 영향을 분석하고 이를 해결하기 위해 오픈소스 기반 컨테이너 시스템인 Docker[2]의 컨테이너 생명 주기(Life-cycle) 관리 기능[3]을 활용한 컨테이너 관리기법을 제안한다. 또한, 컨테이너에서 실행되는 GPU 작업을 개별적으로 추적하기 위한 GPU 작업 모니터링 기법을 설명한다. 본 논문의 주요 기여는 다음과 같다.

- 본 논문에서는 다수의 사용자가 GPU를 공유하는 컨테이너 기반 클라우드 환경에서 자원 경쟁으로 인한 성능 저하를 방지하기 위한 컨테이너 관리기법을 제안한다.
- 본 논문에서 제안한 기법은 사용자 컨테이너 및 GPU 작업과 완전하게 독립적으로 작동하며, 컨테이너 사용자는 자원 경쟁을 완화하기 위한 어떠한 기능도 추가할 필요 없다.
- 본 논문에서 제안한 기법은 컨테이너 외부에서 컨테이너의 GPU 작업 실행 여부를 추적하고 다수의 컨테이너 사이에 GPU 자원 경쟁 상태를 관리한다.
- 실험을 통해 기존 컨테이너 기반 클라우드 환경에서 GPU 자원 경쟁으로 인한 문제를 분석하고 성능 평가를 통해 제안하는 기법의 효율성을 검증한다.

본 논문의 2장은 본 논문에서 제안하는 기법을 설명하기 위한 기반 기술을 설명하며, 3장에서는 실험을 통해 다수의 컨테이너가 GPU를 공유하는 환경에서 자원 경쟁으로 인해 발생하는 성능 영향을 분석한다. 4장에서는 GPU 자원 경쟁으로 인한 성능 영향을 완화하기 위해 본 논문에서 제안하는 컨테이너 관리기법에 관해 설명하고 5장에서는 실험을 통해 본 논문에서 제안한 기법의 효율성을 증명하며, 6장에서는 관련 연구를 설명한다. 마지막 7장에서는 결론 및 향후 연구를 설명한다.

2. 기반 기술

2.1 GPU 프로그래밍 모델과 다중 GPU 프로세스

GPU는 수천 개의 연산 코어를 내장하고 있기 때문에 GPGPU(General-Purpose computing on Graphics Processing Units) 프로그래밍[4] 기술을 활용하여 대규모 병렬 처리(Massively Parallel Processing)를 가능하게 한다. GPU를 활용해 작업을 처리하면 대규모 병렬 처리 기술을 통해 수천~수만 개의 스레드를 사용할 수 있으며, 이를 통

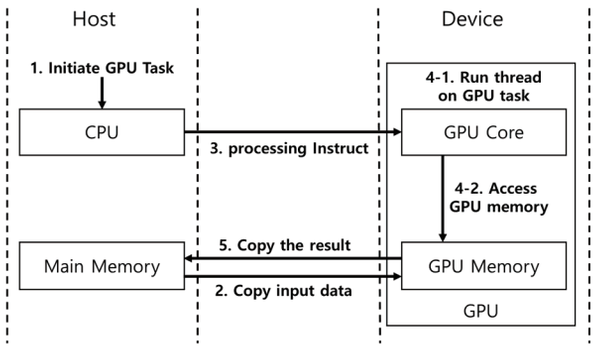


Fig. 1. GPGPU Programming Model

해 고성능 연산을 지원한다. GPU는 CPU의 보조 프로세서(Co-processor)로 작동한다. Fig. 1과 같이 GPU 작업이 실행될 때 CPU에서 처리되는 호스트(Host) 영역과 GPU에서 처리되는 디바이스(Device) 영역에서 처리되며 상호 데이터 전송을 통해 전체 작업을 협업한다[5].

Fig. 1과 같이 GPU 연산 코어는 연산에 사용될 데이터를 GPU 메모리에서 읽어온다. GPU 연산 코어는 데이터를 읽어올 때 GPU 메모리에만 접근할 수 있어서 GPU 연산을 시작하기 위해서는 연산에 사용될 데이터가 GPU 메모리에 존재해야 한다. 하지만 GPU 메모리에 데이터를 직접 작성하거나 읽을 수 없기 때문에 메인 메모리에 데이터를 먼저 작성한 뒤 GPU 메모리로 복사하는 GPU 메모리 입력 작업을 수행하며, GPU 연산 결과 또한 GPU 메모리에 존재하는 연산 결과를 메인 메모리로 출력하는 GPU 메모리 출력 작업이 수행되어야 GPU 작업 결과를 사용할 수 있다.

GPU는 다중 GPU 프로세스를 지원하기 위한 기술을 제공한다. 본 논문에서 사용한 NVIDIA GPU의 경우 다중 GPU 프로세스 환경에서 GPU 작업의 성능 향상을 위해 MPS(Multi-Process Service)[6]라는 기술을 제공한다. 다수의 GPU 작업이 GPU에서 실행될 때 각 GPU 작업은 컨텍스트 저장공간과 스케줄링 자원을 할당받는다[6]. 이로 인해 GPU 프로세스를 동시에 실행할 때 스케줄링 자원을 교환하는 작업으로 인한 오버헤드가 발생한다. 이로 인해 다중 GPU 프로세스가 동시에 실행되면 성능이 저하되는 문제가 발생한다. 하지만 MPS 기술을 활용하면 다중 GPU 프로세스는 단일 컨텍스트로 관리되어 각 GPU 프로세스의 컨텍스트 저장공간과 스케줄링 유닛이 하나의 스케줄링 자원으로 공유해서 다중 GPU 프로세스가 독립적으로 실행되는 환경과 비교하였을 때 컨텍스트 교환으로 인한 오버헤드가 감소한다.

MPS 기술은 단일 사용자로부터 실행된 다수의 GPU 작업 사이에서만 사용 가능하며, 본 논문에서 사용한 컨테이너 환경에서는 특정 GPU 제품을 제외하고는 활용할 수 없는 제한이 있다. 따라서, 본 논문에서는 GPU 프로세스에 컨텍스트 저장공간과 스케줄링 리소스가 독립적으로 할당되는 다중 GPU 프로세스 환경에서 자원 경쟁으로 인한 성능 저하를 방지하기 위한 컨테이너 관리기법을 제안한다.

2.2 컨테이너 환경에서의 GPU 공유

GPU 작업은 앞서 설명한 일련의 과정을 통해 수행되며 클라우드 환경에서 GPU는 컨테이너가 GPU 작업을 수행하지 않으면 사용되지 않는다. 이로 인해 GPU가 공유되지 않는 환경에서는 GPU를 할당받은 컨테이너가 GPU 작업을 수행하지 않으면 GPU는 유휴 상태가 되며, 자원 활용률을 낮추고 GPU가 활용되지 않는 시간으로 인한 비용 낭비 문제를 발생시킨다.

본 논문에서 활용한 컨테이너 시스템인 Docker는 GPU 관리 도구인 NVIDIA Docker[7]를 통해 컨테이너가 GPU를 활용할 수 있도록 지원하며, 다수의 컨테이너가 단일 GPU에 접근하는 것을 허용한다. 이를 통해 다수의 컨테이너에서 실행되는 GPU 작업은 단일 GPU에서 다중 프로세스 형태로 동시에 실행될 수 있다. GPU를 단일 컨테이너가 독점하는 경우 해당 컨테이너가 GPU 작업을 실행하지 않는 상황에서 GPU는 활용되지 않고 유휴 상태가 되며, 컨테이너에서 실행한 GPU 작업이 GPU 자원을 100% 사용하지 않는다면 유휴 자원이 발생한다. GPU가 공유되지 않는 경우에 발생하는 GPU 유휴 상태는 단순하게 자원 활용률 측면의 문제뿐만 아니라 유휴 상태인 자원을 다른 컨테이너도 사용하지 못하는 문제를 발생시킨다. GPU를 다른 컨테이너가 활용하지 못한다면 GPU 작업을 실행하는 컨테이너마다 GPU를 독립적으로 할당해야 한다는 것을 뜻하며, 서버에 설치할 수 있는 GPU의 개수는 제한적이기 때문에 단일 서버에 호스팅 할 수 있는 컨테이너의 개수도 줄어들게 된다.

하지만, 단일 GPU를 무조건 많은 수의 컨테이너가 공유되도록 하는 것은 바람직하지 않다. 다수의 컨테이너가 GPU를 공유하는 환경은 필연적으로 자원 경쟁 문제를 동반한다. GPU 자원의 공유는 자원 활용률을 증가시키고 유휴 시간을 감소하는 장점이 있지만, GPU를 공유하는 컨테이너의 개수가 GPU 자원 허용 범위를 넘어서면 자원 경쟁이 심해지며, 이는 성능 저하로 이어진다. 따라서, 다수의 컨테이너가 단일 GPU를 공유하는 환경에서 성능 저하를 방지할 수 있도록 자원 경쟁 상태를 관리해야 한다.

본 논문에서는 앞서 설명한 GPU 작업의 특성과 컨테이너 생명 주기 관리 기능을 활용하여 GPU 자원 공유로 인한 자원 경쟁을 완화하고 성능 저하를 방지하기 위한 컨테이너 관리기법을 제안한다. 다음 장에서는 실험을 통해 다수의 컨테이너가 GPU를 공유하는 컨테이너 기반 클라우드 환경에서 GPU 공유로 인한 자원 경쟁과 그에 따른 성능 저하 문제를 분석한다.

3. GPU 자원 공유로 인한 성능 영향

이번 장은 다중 GPU 프로세스 환경에서 GPU 작업의 성능을 측정하고 자원 경쟁으로 인한 성능 저하 문제를 분석한다. 앞서 설명한 것과 같이 GPU는 다중 프로세스 환경을 지원한다. 다수의 GPU 작업이 실행되면 GPU 내부의 하드웨어

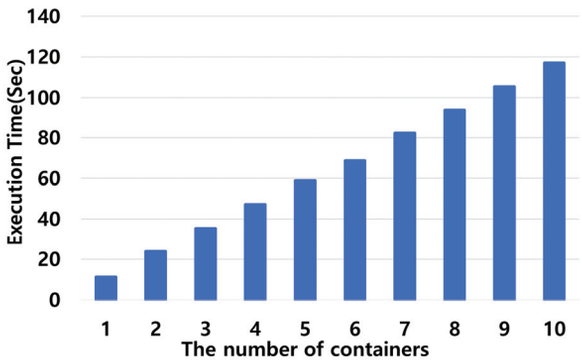


Fig. 2. Performance of 13,312×13,312 Matrix Multiplication

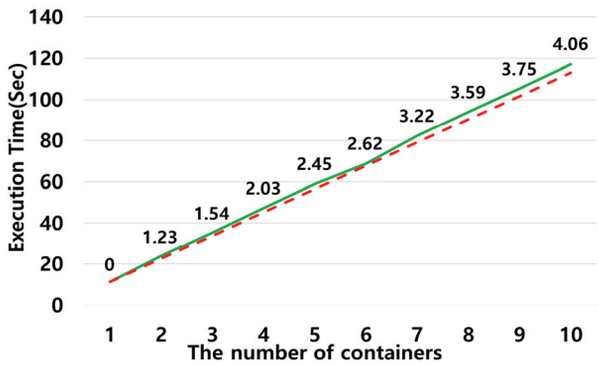


Fig. 3. Performance Degradation Owing to Concurrent GPU Tasks

스케줄러를 통해 GPU 작업이 스케줄링 되며, 작업 사이에서 컨텍스트 전환을 통해 다수의 작업이 GPU에서 동시에 처리될 수 있도록 지원한다.

다중 GPU 작업의 GPU 공유를 통해 단일 프로세스 환경과 비교했을 때 상대적으로 GPU 자원의 활용률을 최대화할 수 있으며, GPU의 컴퓨팅 능력을 충분히 활용할 수 있다. 하지만 다수의 프로세스가 GPU를 공유하게 되면 독립적인 GPU 작업 사이에서 스케줄링 오버헤드가 발생한다. 또한, 다수의 GPU 작업이 동시에 실행되면 자원 경쟁에 대한 문제도 발생한다. 이번 장에서 수행할 실험은 다수의 컨테이너가 GPU 작업을 동시에 실행할 때 GPU 작업의 성능을 측정하여 다중 GPU 작업이 동시에 실행되는 환경에서의 성능 저하 문제를 분석한다. 실험에서는 GPU 메모리 용량이 24GB인 NVIDIA GPU를 사용하며, 각 컨테이너는 CUDA로 구현된 13,312×13,312 행렬 곱셈 연산을 수행한다. GPU 작업을 실행하는 각 컨테이너는 약 2.2GB의 GPU 메모리를 사용하며, 10개의 컨테이너가 총 24GB의 GPU 메모리 중 약 23GB의 메모리를 사용하기 때문에 10개의 컨테이너를 사용할 때 실행 가능한 가장 큰 규모의 행렬 곱셈 연산을 수행하게 된다. 실험 결과는 Fig. 2와 같으며, GPU 작업이 시작되고 모든 GPU 작업이 완료된 시간을 측정한다.

Fig. 2의 실험 결과에서 보여주는 것과 같이 GPU 작업을 실행하는 컨테이너의 개수가 증가할수록 GPU 작업의 실행 시간이 증가하며, 다중 GPU 프로세스의 동시 실행으로 인한 성능 저하는 Fig. 3에서 보여준다.

Fig. 3은 다수의 GPU 작업을 하나씩 차례대로 실행하여 다중 GPU 프로세스로 인한 자원 경쟁이 없는 상태와 GPU 작업을 동시에 실행해 자원 경쟁이 발생한 경우의 성능을 비교한다. Fig. 3에서 실선은 GPU 작업을 동시에 실행하였을 때 GPU 작업의 실행 시간을 나타낸다. 점선은 GPU 작업의 단일 성능을 기준으로 $y = ax$인 가상의 기준선을 나타내며, a는 13,312×13,312 행렬 곱셈의 단일 실행 시간이다. 즉, Fig. 3에서 점선으로 표시된 기준선은 GPU 작업을 하나씩 차례대로 처리한 경우의 성능으로 볼 수 있다. 그리고 선형 그래프 위의 숫자는 GPU 작업을 동시에 실행했을 때의 성능과 기준선 값 사이의 차이를 나타낸다.

Fig. 3의 실험 결과에서 보여주는 것과 같이 다수의 컨테이너에서 GPU 작업을 동시에 실행하면 다수의 GPU 작업을 단일 GPU에서 스케줄링하기 위한 추가 실행 시간이 발생한다. GPU 작업의 성능과 기준선을 비교하면 2개의 컨테이너가 GPU 작업을 동시에 실행했을 때 성능 차이가 1.23초이지만 10개의 컨테이너가 GPU 작업을 실행하였을 때 성능 차이가 4.06초까지 증가한다. 이는 GPU 작업을 동시에 처리했을 때 발생하는 성능 저하를 보여준다. 이처럼 GPU 작업을 동시에 실행하게 되면, 자원 경쟁으로 인한 성능 저하가 발생한다.

Fig. 2를 통해 설명한 GPU 작업의 전체 성능에서 GPU 메모리 입력, Kernel 함수 실행 그리고 GPU 메모리 출력 작업이 실행되는 각각의 시간은 Table 1에서 보여준다.

Table 1에서 보여주는 것과 같이 GPU 작업을 실행하는 컨테이너의 개수가 증가할수록 GPU 메모리 입출력 작업 실행 시간이 증가한다. 하지만, GPU 메모리 입출력 작업은 GPU 작업의 전체 성능에 미치는 영향이 매우 작으며, GPU 연산 작업, 즉, Kernel 함수의 실행 시간이 전체 성능에 큰 영향을 미치는 것을 볼 수 있다. 실험에서 각 컨테이너에서 실행하는 GPU 행렬 곱셈 작업은 각 항을 계산하기 위해 총 177,209,344개의 스레드를 생성하며, GPU 코어를 100% 사용하게 된다. CUDA를 사용한 GPU 작업은 앞서 설명한

Table 1. Experiment Environment

Number of Container	GPU Memory Input time(Sec)	Kernel Processing time(Sec)	GPU Memory Output time(Sec)
1	0.188997	10.79455	0.318502
2	0.235931	23.27129	0.332669
3	0.333234	34.78156	0.340334
4	0.471171	46.41933	0.348032
5	0.588509	58.01878	0.361234
6	0.698254	69.37386	0.362889
7	0.808317	81.16589	0.364612
8	0.933156	92.69842	0.382235
9	1.042857	104.0067	0.42654
10	1.142859	115.4842	0.462421

것과 같이 각 프로세스에 스케줄링 리소스가 할당되며, 이로 인해 다수의 컨테이너에서 GPU 작업을 동시에 실행하게 되면 GPU 프로세스 사이에서 스케줄링 리소스를 교체하는 컨텍스트 스위칭[6]이 발생하고 GPU를 공유하는 컨테이너가 증가할수록 이로 인한 추가 실행 시간이 증가한다.

GPU 작업이 동시에 실행되는 경우 모든 GPU 작업은 GPU의 하드웨어 스케줄러에 의해 GPU를 공유하며 동시에 처리된다. 다수의 GPU 작업이 GPU에서 동시에 처리될 때 특정 컨테이너의 GPU 작업이 지연되어 전체 성능이 저하되는 것이 아니라 자원 경쟁으로 인해 모든 GPU 작업의 실행 시간이 균일하게 증가한다. 앞서 수행한 실험에서 각 행렬 곱셈을 실행하는 컨테이너의 개수가 최대일 때 각 컨테이너에서 실행하는 GPU 작업의 개별 성능은 Fig. 4에서 보여준다. Fig. 4는 10개의 컨테이너에서 13,312×13,312 행렬 곱셈 작업을 동시에 실행했을 때의 각 컨테이너의 개별 성능을 보여준다.

Fig. 4의 실험 결과와 같이 모든 GPU 작업의 성능은 1% 이하의 성능 차이만 발생시키고 균등하게 처리되며, GPU 자원 경쟁으로 인해 모든 컨테이너의 GPU 작업이 균일하게 성능 영향을 받게 된다. 앞서 설명한 것과 같이 GPU 작업을 수행할 때 GPU 메모리에 데이터 전송을 완료한 후 Kernel 함수를 실행한다. 또한 실험 결과에서 보듯이 Kernel 함수의 실행 시간이 GPU 작업을 차례대로 실행한 경우보다 성능이 저하된 문제도 Kernel 함수의 실행 시간의 증가로 인한 영향 때문이다.

다수의 컨테이너가 GPU를 공유할 때 Kernel 함수의 실행 시간이 증가하는 문제는 앞서 수행한 실험의 결과를 통해 설명한 것과 같이 다수의 GPU 작업이 단일 GPU에서 동시에 실행될 때 GPU 연산 코어 경쟁으로 인한 문제이다. 특히, GPU 작업의 동시 실행으로 인한 리소스 교체 작업으로 인해 GPU 작업이 실행될 때 추가 실행 시간 발생하며, 이로 인해 실행 중인 모든 GPU 작업의 성능에 영향을 미치게 된다.

본 논문에서는 앞서 수행한 실험을 통해 확인한 것과 같이 GPU 작업이 동시에 실행될 때 자원 경쟁으로 인해 발생하는 성능 저하를 최소화하기 위한 컨테이너 관리기법을 제안한다. 본 논문에서 제안하는 컨테이너 관리기법은 다음 장에서 자세히 설명한다.

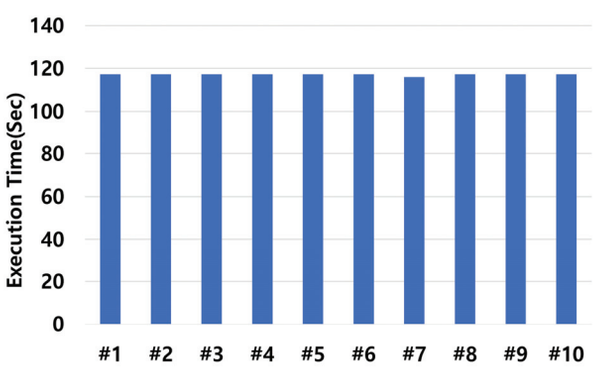


Fig. 4. Individual Performance of GPU Tasks

4. GPU 자원 경쟁을 완화하기 위한 컨테이너 관리기법

본 논문은 앞서 설명한 것과 같이 컨테이너 기반 클라우드 환경에서 다수의 컨테이너의 GPU 작업이 동시에 실행할 때 발생하는 자원 경쟁 상태를 관리하기 위한 컨테이너 관리기법을 제안한다. 본 논문에서 제안하는 기법은 크게 2가지 서브 시스템으로 구성된다. 첫 번째 서브 시스템은 GPU 작업을 수행하는 컨테이너와 GPU 자원 사용량을 추적하기 위한 모니터링 모듈이며, 두 번째 서브 시스템은 GPU 자원 사용량에 따라 컨테이너의 GPU 작업 실행 여부를 결정하는 컨테이너 관리 모듈이다.

본 논문에서는 다수의 GPU 작업이 동시에 실행될 때 자원 경쟁을 방지하기 위해 GPU 자원의 사용량에 따라 GPU 작업을 시작할 컨테이너를 결정한다. GPU 자원이 많이 사용되는 경우 GPU 작업을 실행하는 컨테이너를 일시 정지하여 GPU 작업의 실행을 지연시키고 자원 경쟁을 방지한다. 본 논문에서 제안하는 컨테이너 관리 시스템의 전체 구조는 Fig. 5와 같다.

4.1 컨테이너의 GPU 작업 실행 상태 추적

자원 경쟁을 방지하기 위해 특정 컨테이너의 GPU 작업의 실행을 지연시키기 위해서는 먼저 서버에서 실행 중인 컨테이너의 GPU 작업 실행 여부를 확인해야 한다. 본 논문에서는 GPU 자원의 경쟁을 방지하기 위해 앞서 설명한 것과 같이 컨테이너에서 실행되는 GPU 작업을 지연시키며, GPU 작업을 지연시키기 위해 *docker pause*, *unpause*와 같이 컨테이너의 일시 정지 기능[3]을 사용한다. GPU 작업은 외부에서 실행 여부를 조절할 수 없기 때문에 본 논문에서는 컨테이너의 일시 정지 기능을 활용해 특정 컨테이너의 GPU 작업을 지연시킨다. 컨테이너의 일시 정지 기능을 통해 GPU 작업이 지연되는 컨테이너는 GPU 작업을 실행하는 컨테이너

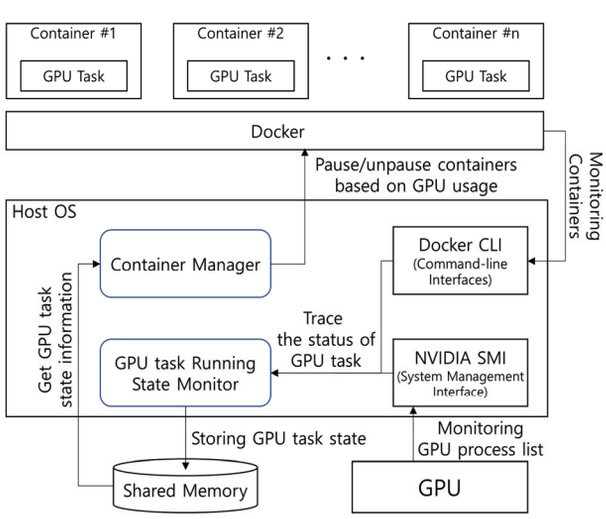


Fig. 5. System Structure

사이에서 결정되며, 이를 위해서는 GPU 작업을 실행하는 컨테이너를 찾아야 한다.

GPU 활용이 가능한 컨테이너 환경은 컨테이너와 GPU를 위한 모니터링 기술이 제공된다. 컨테이너 모니터링 기술[8,9]은 서버에서 실행되는 컨테이너와 관련된 정보를 추적하며, GPU 모니터링 기술[10]은 GPU 자원 사용량과 GPU 작업에 대해 추적한다. 하지만, 본 논문에서 컨테이너 기반 클라우드 환경으로 사용한 Docker의 경우 GPU 장치를 관리하기 위해 NVIDIA Docker를 사용하며, NVIDIA SMI (System Management Interface)를 통해 모니터링되기 때문에 컨테이너와 GPU에 대한 관리 도구가 분리되어있다. 이로 인해 컨테이너 모니터링 기술에서는 컨테이너와 컨테이너에서 실행 중인 프로세스 목록에 대한 정보를 얻을 수 있지만, GPU 작업에 대한 모니터링 정보를 얻을 수 없고 GPU 모니터링 기술에서는 GPU 작업 목록과 GPU 자원 사용량에 대한 정보는 얻을 수 있지만, GPU 작업을 실행하는 컨테이너에 대한 정보를 얻을 수 없다. 본 논문에서는 GPU 자원 사용량, 실행 중인 GPU 작업의 목록과 컨테이너 목록에 대한 정보를 통합하여 GPU 작업이 실행 중인 컨테이너를 찾는다. GPU 작업을 실행하는 컨테이너를 추적하기 위한 모니터링 기법은 Algorithm 1과 같이 작동한다.

Algorithm 1에서 보여주는 것과 같이 본 논문에서는 GPU 작업을 실행하는 컨테이너를 찾기 위해 GPU 모니터링 정보에서 GPU 작업 프로세스가 발견되면 GPU 작업 프로세스 ID와 각 컨테이너에서 실행 중인 프로세스 ID를 비교한다. GPU 작업에서 GPU 연산 작업을 수행할 때만 실제로 GPU가 사용되기 때문에 GPU 작업을 실행하는 컨테이너를 찾은 후에는 해당 컨테이너가 실제 GPU를 사용하는지 찾아야 한다.

앞서 설명한 것과 같이 GPU 작업은 GPU 작업은 호스트와 디바이스 영역으로 구분되며, GPU 작업에서 GPU는 디바이스 영역에서만 사용된다. 또한, GPU를 사용한 연산 작업

Algorithm 1: GPU task state monitor

```
1  while()
2    get GPU process list;
3    if num of GPU process 〉 0
4      Get active Container list
5        for 0 ~ num of Container
6          Get Container's process list
7            if Container's process ID
                       == GPU process ID
8              struct GPUtaskinfo[] =
                 {Container ID, GPU process ID ... ,
                 GPUmemory usage, GPUcore usage}
9    if GPUtaskinfo[] != Previous GPUtaskinfo
10     SharedMemory = GPUtaskinfo[]
```

전후로 메인 메모리와 GPU 메모리 사이에 데이터 전송 작업이 발생하게 되는데 본 논문에서는 이러한 GPU 작업의 특성을 활용하여 해당 컨테이너에서 실행 중인 GPU 프로세스의 GPU 메모리 사용량이 0 이상일 때 GPU 메모리 입력 작업을 수행 중이라고 판단하고 이를 통해 GPU 작업을 실행하는 컨테이너가 실제 GPU를 사용하고 있다고 판단하며, GPU 메모리가 사용되다가 GPU 메모리 사용량이 0이 되면 GPU 메모리 출력 작업이 완료된 것으로 보고 해당 컨테이너의 GPU 사용이 완료되었다고 판단한다.

앞서 설명한 GPU 작업 실행 상태 추적 모니터는 지속적으로 GPU 작업의 상태를 추적한다. GPU 작업 실행 상태를 수집할 때 다른 작업에 의해 방해받지 않기 위해 독립적인 프로세스에서 작동하며, 수집된 GPU 작업상태 정보는 공유 메모리에 저장되고 GPU 작업상태에 변동 사항이 있을 때 공유 메모리에 저장된 정보를 업데이트한다.

본 논문에서 제안한 컨테이너 관리기법은 GPU 작업 실행 상태 모니터에서 수집한 GPU 작업 실행 상태와 GPU 자원 사용량을 기반으로 GPU 작업의 시작을 지연시켜 GPU 자원의 경쟁으로 인한 성능 저하를 최소화한다. GPU 자원 경쟁을 방지하기 위한 컨테이너 관리기법은 다음 장에서 자세히 설명한다.

4.2 GPU 자원 경쟁을 방지하기 위한 컨테이너 관리

앞서 설명한 것과 같이 본 논문에서는 다수의 컨테이너가 GPU 작업을 동시에 실행할 때 GPU 자원 사용량을 기반으로 Docker의 컨테이너 생명 주기(Life-Cycle)를 관리하는 *docker pause*, *unpause*를 사용해 컨테이너를 일시 정지시켜 GPU 작업의 실행을 지연시키고 GPU 자원 경쟁 상태를 방지한다.

다수의 컨테이너가 GPU 작업을 동시에 실행할 때 자원 경쟁을 방지하기 위해서는 작업을 실행할 컨테이너와 일시 정지할 컨테이너를 결정해야 한다. 이를 위해서는 서버에서 실행 중인 다수의 컨테이너 중에서 GPU 작업을 실행하는 컨테이너에 대한 정보와 컨테이너에서 실행한 GPU 작업의 실행 순서에 대한 정보는 중요하다.

GPU 작업을 실행하는 컨테이너를 찾는 작업은 앞서 설명한 컨테이너의 GPU 작업 실행 상태 추적 기법을 통해 컨테이너에서 실행하는 GPU 작업의 시작 및 종료 시점을 찾을 수 있기 때문에 이를 통해 GPU 프로세스를 실행한 컨테이너를 특정하고 어떤 컨테이너가 어떤 GPU 작업을 실행하고 있는지 찾을 수 있다. 또한, 앞서 설명한 것처럼 본 논문에서는 GPU 작업을 실행하는 컨테이너를 찾을 때 프로세스 ID를 기반으로 GPU 작업을 추적한다. GPU 작업의 프로세스 ID는 실행 순서에 따라 오름차순으로 할당되기 때문에 프로세스 ID를 통해 실행 순서를 확인할 수 있다.

본 논문에서 GPU 작업을 처리할 때 기본적으로 FIFO 기반으로 처리한다. 먼저 실행된 순서대로 GPU를 사용하도록

조절하고 GPU 자원이 부족해지면 GPU 작업을 상대적으로 나중에 실행한 컨테이너를 일시 정지시킨다. Algorithm 2는 본 논문에서 제안한 컨테이너 관리기법의 작동 방식을 보여준다.

Algorithm 2과 같이 본 논문에서 제안한 컨테이너 관리기법은 앞서 설명한 GPU 작업 실행 상태 모니터를 통해 수집되어 공유 메모리에 저장된 정보를 기반으로 작동한다. 컨테이너 관리 알고리즘은 초기 상태에 GPU 작업 상태 모니터의 공유 메모리가 업데이트될 때까지 대기한다. 공유 메모리에 새로운 모니터링 정보, 즉, GPU 작업이 발견되면 컨테이너 관리 작업을 시작한다. 컨테이너 관리 알고리즘은 line 3, 13과 같이 현재 실행 중인 GPU 작업의 존재 여부에 따라 작동한다. 현재 실행 중인 GPU 작업이 없다면 GPU 프로세스 목록의 가장 첫 번째 GPU 작업을 실행하는 컨테이너는 놔두고 두 번째 컨테이너부터 일괄적으로 일시 정지한다. 컨테이너의 GPU 자원 사용량을 미리 알 수 없기 때문에 FIFO 방식으로 가장 처음 GPU 작업을 실행한 컨테이너의 GPU 작업을 실행시키고 두 번째 컨테이너부터 일괄적으로 일시 정지 시킨다. 그리고 현재 GPU 코어 사용량이 100% 미만이라면 컨테이너의 일시 정지를 하나씩 해제하면서 현재 GPU 코어 사용량이 100%가 되면 나머지 컨테이너의 일시 정지 상태를 유지한다. 그리고 1개 이상의 GPU 작업이 이미 실행 중이라면 GPU 코어의 사용량을 확인하여 새로 실행된 GPU 작업의 일시 정지 여부를 결정한다.

Algorithm 2: Container management

```
1  while()
2    if be updated SharedMemory
3      if num of running task != 0
4        if GPU Core usage == 100%
5          for num of running task + 1
                              ~ num of Container
6            pause container
7        else
8          while( GPU Core usage 〈 100% )
9            unpause container
10           num of running task++
11           if GPU memory usage 〉 95%
12             pause last unpaused container
13     else if number of running task == 0
14       num of running task++
15       for 1 ~ num of Container
16         pause container
17         if GPU Core usage 〈 100%
18           while( GPU Core usage 〈 100% )
19             unpause container
20             num of running task++
21             if GPU memory usage 〉 95%
22               pause last unpaused container
```

논문에서 제안하는 기법은 기본적으로 현재 GPU 코어가 100%라면 새로 GPU 작업을 시작한 컨테이너는 모두 일시 정지시킨다. 또한, CUDA를 사용한 GPU 작업에서 GPU 메모리를 할당하는 *cudaMalloc()* 함수는 GPU 메모리의 초과 사용을 허용하지 않기 때문에 본 논문에서도 현재 GPU 메모리가 100%가 되기 전에 가용 GPU 코어가 존재해도 가장 마지막에 시작된 GPU 작업을 실행하는 컨테이너를 일시 정지한다. 본 논문에서는 GPU 메모리가 95% 이상 사용되면 가장 마지막에 작업을 시작한 컨테이너를 일시 정지하도록 구현하였다.

GPU 작업은 앞서 설명한 것과 같이 GPU 연산 작업이 실행되면 블랙박스 형태로 처리되며, GPU에서 실행될 코드가 GPU에 전달되고 GPU 코어를 사용한 연산 작업이 시작되면 컨테이너를 일시 정지해도 GPU 연산은 계속 진행된다. 이로 인해 Fig. 6에서 보여주는 것과 같이 CPU 작업 구간이나 GPU 메모리 입력 단계와 같이 GPU 코어를 사용하기 전의 호스트 영역에서 작업을 처리할 때 컨테이너를 일시 정지시켜야 GPU 작업을 일시 정지할 수 있다.

컨테이너의 일시 정지를 통해 GPU 작업을 일시 정지하기 위해서는 호스트 영역의 작업 중 GPU 메모리 입력 단계에서 컨테이너 일시 정지 작업을 수행해야만 한다. 호스트 영역에서 일반적인 CPU를 사용한 작업에서는 GPU를 사용하지 않아서 일시 정지할 필요가 없기 때문에 본 논문에서는 GPU 작업이 GPU 코어를 사용하기 바로 이전 단계인 GPU 메모리 입력 단계에 컨테이너를 일시 정지하여 GPU 작업을 지연시킨다.

컨테이너에서 실행 중인 GPU 작업의 GPU 메모리 입력 작업이 완료된 직후에 컨테이너를 일시 정지하면 일시 정지 해제 후 즉시 GPU 연산을 수행할 수 있다. 하지만 컨테이너가 GPU 작업을 실행할 때 각 컨테이너가 GPU 메모리에 전송할 입력 데이터의 용량은 미리 알 수 없다. GPU 메모리에 데이터 입력이 완료된 후에 해당 컨테이너의 입력 데이터 용량을 알 수 있으며, 일반적인 컨테이너 환경에서 GPU 메모리 사용량을 제한할 수 없기 때문에 GPU 메모리 입력 작업이 완료되고 GPU 코어를 사용한 Kernel 함수를 시작하기 직전에 GPU 작업을 일시 정지하는 것은 거의 불가능하다. GPU 작업 자체에 작업 지연을 위한 코드를 추가하거나 컨테이너 사용자에게 GPU 작업에 대한 정보를 요청할 수도 있지만 이러한 사용자 측면에 의존한 방법들은 GPU 작업의 프로그램 코드를 수정해야 해서 투명성을 훼손하고 자원 사용 정보를 사용자에게만 의존해야 해서 신뢰성 측면의 문제가 발생할 수 있다.

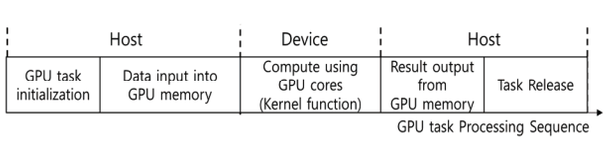


Fig. 6. Host and Device Areas for the GPU Task

따라서, 본 논문에서는 컨테이너에서 실행된 작업 중 가장 첫 번째 실행된 GPU 작업을 우선 실행하고 두 번째 실행된 작업부터 GPU 메모리 입력 작업이 시작되어 GPU 메모리 사용량이 0 이상일 경우 GPU 작업이 시작되었다고 판단하여 컨테이너를 일시 정지한 후 GPU 자원 사용량을 기반으로 일시 정지를 해제하여 이후 일시 정지 중인 다른 컨테이너의 GPU 작업을 수행하게 한다. GPU 자원이 모두 사용되어 일시 정지된 작업이 존재하는 경우 작업을 실행 중인 컨테이너의 GPU 메모리 사용량을 확인하여 GPU 메모리 사용량이 0이 되면 다음 작업부터 차례대로 실행한다. 이를 통해 이전 장에서 수행한 실험에서 GPU 자원 경쟁으로 인한 성능 저하를 방지한다.

본 논문에서는 앞서 설명한 GPU 작업 실행 상태 추적 기법과 컨테이너 관리기법을 통해 컨테이너에서 실행된 GPU 작업의 실행 상태를 추적하고 GPU 자원 사용량에 따라 GPU 작업 실행 순서에 맞춰 GPU 작업을 실행하는 컨테이너를 일시 정지시켜 GPU 작업의 실행을 지연시킨다. GPU 작업의 실행 상태를 추적하기 위해 사용한 NVIDIA-SMI의 경우 GPU의 상태와 GPU 프로세스 목록을 조회하는 데 약 0.06초가 소요되며, 컨테이너의 일시 정지와 일시 정지 해제는 모두 약 0.06초가 소요되기 때문에 매우 짧은 주기로 GPU 작업을 추적하고 컨테이너를 일시 정지하여 GPU 자원 경쟁을 방지할 수 있다.

본 논문에서 제안한 기법은 일시 정지할 수 없는 Kernel 함수 실행 단계가 되기 전에 GPU 메모리 입력 단계에서 컨테이너를 일시 정지시켜 GPU 코어의 경쟁을 방지한다. 다음 장에서는 실험을 통해 본 논문에서 제안한 컨테이너 관리기법의 효율성을 검증하기 위해 GPU 작업의 성능을 측정한다.

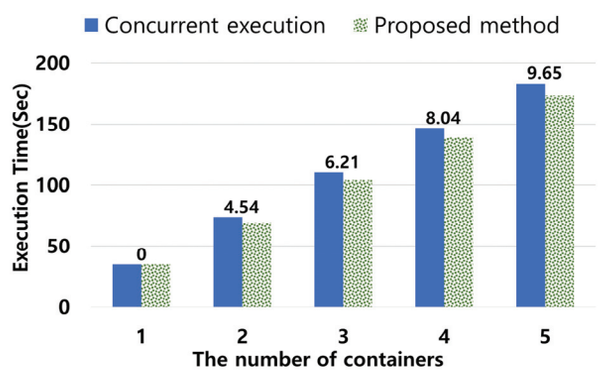
5. 실 험

이번 장에서는 본 논문에서 제안한 컨테이너 관리기법의 효율성을 검증하기 위한 실험을 수행한다. 실험에서는 다수의 컨테이너가 GPU 작업을 동시에 실행하는 컨테이너 기반 클라우드 환경에서 GPU 작업의 성능 측정을 통해 본 논문에서 제안한 컨테이너 관리기법이 GPU 자원 경쟁을 효율적으로 방지하고 자원 경쟁으로 인한 성능 저하를 완화할 수 있는지 확인한다. 실험에서 사용된 컨테이너는 1개의 GPU를 공유하며, 각 컨테이너는 모두 같은 작업을 독립적으로 수행하며, 동시에 실행한다. 실험에서 사용한 클라우드 서버의 성능은 Table 2와 같다.

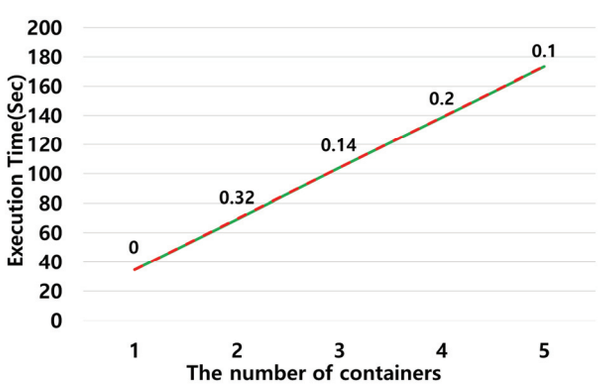
Table 2. Experiment Environment

	Container Server
CPU	i9-10920X (3.5GHz)
Memory	128 GiB
GPU	RTX 3090 (24GiB Memory)
OS	Ubuntu 18.04
Docker	NVIDIA Docker2

실험에서는 다수의 컨테이너를 사용하여 다양한 크기의 행렬 곱셈 작업을 수행한다. 동시에 GPU 작업을 실행하는 컨테이너는 모두 동일한 작업을 수행하며, 행렬 크기에 따라 GPU 작업을 실행하는 컨테이너의 개수를 다르게 실험을 수행했다. 실험은 컨테이너를 최대 5개, 10개를 사용해 각각 19,456×19,456과 13,312×13,312 크기의 행렬 곱셈 작업을 수행하며, GPU 작업을 동시에 실행하는 컨테이너를 하나씩 증가시키며 성능을 측정한다. 실험에 사용한 컨테이너의 개수와 행렬 곱셈 작업의 크기는 가용 GPU 메모리 용량을 고려해 실행 가능한 가장 큰 규모의 행렬 곱셈 작업을 수행한다. 실험 결과는 Fig. 7과 8에서 보여주며, Fig. 7(a)와 8(a) 그래프의 막대 위 값은 GPU 작업이 GPU를 공유하여 동시에 처리하는 기존 환경과 본 논문에서 제안한 컨테이너 관리기법을 적용했을 때의 성능 차이를 보여준다. Fig. 7(b)와 8(b)는 이전 장에서 수행한 실험과 유사하게 가상의 기준선과 본 논문에서 제안한 기법을 적용했을 때의 성능을 비교한다. 기준선은 실선으로 표시되며, 그래프 위의 값은 본 논문의 제안한 기법을 적용했을 때의 성능과 기준선의 실행 시간 차이를 보여주며, 값이 클수록 기준선보다 성능이 좋다는 것을 나타낸다. 실험 결과는 이전 장에서 수행한 실험과 같이 GPU 작업이 시작되고 모든 GPU 작업이 완료될 때까지의 시간을 측정한다.

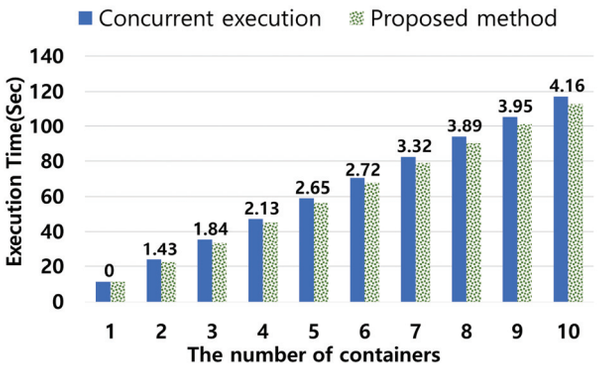


(a) Performance of proposed method

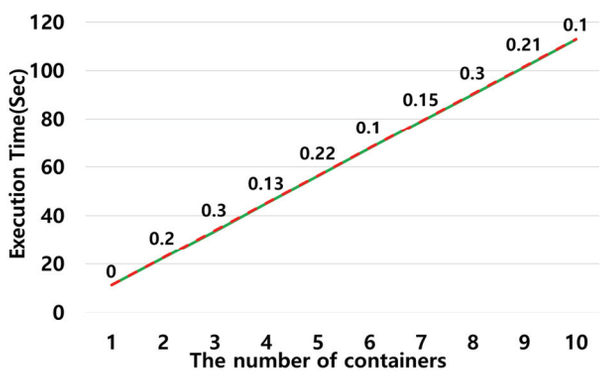


(b) Performance of baseline and the proposed technique

Fig. 7. Improve Performance of 19,456×19,456 Matrix Multiplication with Proposed Container Management Techniques

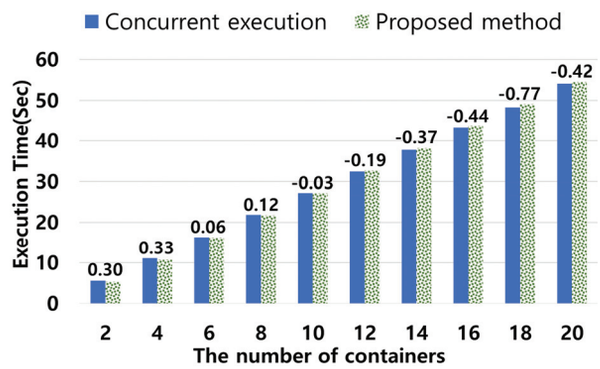


(a) Performance of proposed method

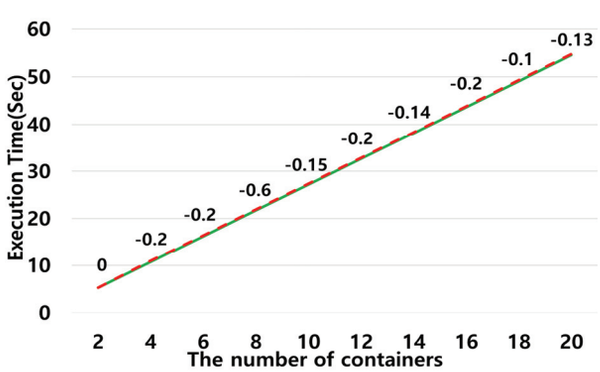


(b) Performance of baseline and the proposed technique

Fig. 8. Improve Performance of 13,312×13,312 Matrix Multiplication with Proposed Container Management Techniques



(a) Performance of proposed method



(b) Performance of baseline and the proposed technique

Fig. 9. Performance for Small GPU Tasks

Fig. 7과 8에서 보여주는 것과 같이 본 논문에서 제안한 컨테이너 관리기법을 적용했을 때의 성능과 비교했을 때 GPU 작업을 동시에 수행하는 컨테이너의 개수가 증가할수록 성능 차이가 증가한다. 특히, 13,312×13,312 행렬 곱셈의 경우 19,456×19,456 행렬 곱셈보다 상대적으로 GPU 메모리 사용량이 적기 때문에 실험에서는 동시에 더 많은 컨테이너를 실행하며, GPU 자원을 공유하는 대상이 더 많아지게 된다. 하지만, 실험 결과에서 보는 것과 같이 자원 경쟁 상대, 즉, GPU를 공유하는 프로세스의 개수보다 GPU 작업의 규모가 자원 경쟁으로 인한 성능 저하에 더 큰 영향을 준다는 것을 알 수 있으며, GPU 작업의 규모가 클수록 GPU 자원을 동시에 실행하는 기존 환경보다 GPU 자원 사용량에 따라 GPU 작업의 실행을 지연시켜 GPU 자원 경쟁을 방지하는 본 논문의 방법이 GPU를 공유하는 환경에서 다중 GPU 작업을 처리할 때 더 효율적이라는 것을 확인할 수 있다.

또한, Fig. 7(b)와 8(b)에서 보여주는 것과 같이 기준선의 성능보다 약 0.2초의 실행 시간이 감소하지만, 전체 실행 시간과 비교하면 큰 성능 향상은 없다. 본 논문에서는 GPU 메모리 입력 작업이 시작되어 GPU 메모리 사용량이 0을 초과하면 GPU 작업이 시작되었다고 판단하기 때문에 초기 상태에 GPU 작업상태 모니터가 GPU 작업을 발견하기 전까지 GPU 메모리 입력 작업이 미리 실행되어 제안된 기법을 사용했을 때 GPU 작업의 전체 실행 시간이 기준선보다 아주 조금 감소하게 된다. 또한, 실험 결과에서 보여주는 것과 같이 다수의 컨테이너가 GPU 작업을 동시에 실행하는 환경에서 GPU 작업을 실행하는 컨테이너의 개수, 즉, GPU에서 실행되는 GPU 작업의 개수보다 GPU 작업의 규모가 더 큰 영향을 미친다는 것을 확인할 수 있다.

Fig. 9의 실험은 앞서 수행한 실험보다 GPU 작업 규모가 상대적으로 더 작은 8,192×8,192 행렬 곱셈 작업을 수행하며, 컨테이너는 최대 20개를 사용한다. Fig. 9에서 보여주는 것과 같이 GPU 작업의 규모가 작으면 GPU 작업을 실행하는 컨테이너가 증가해도 다중 GPU 작업의 동시 실행으로 인한 추가 실행 시간이 작게 발생한다. 이로 인해 본 논문에서 제안한 컨테이너 관리기법을 적용했을 때의 성능 향상 정도가 낮게 측정되었으며, 성능이 더 안 좋아진 경우도 발생했다. 하지만 제안한 컨테이너 관리기법으로 인한 성능 저하 수준은 전체 실행 시간에서 미미한 정도를 차지하며, 앞서 수행한 실험과 같이 규모가 큰 GPU 작업의 경우에는 효율적으로 작동하는 것을 확인할 수 있다. 실험에서 상대적으로 작은 규모의 GPU 작업에 대한 처리는 추후 연구를 통해 해결할 계획이다.

이번 장에서 수행한 실험을 통해 본 논문에서 제안한 컨테이너 관리기법은 각 컨테이너에서 실행하는 GPU 작업이 동시에 실행되는 환경에서 GPU 자원 경쟁으로 인한 성능 저하를 효율적으로 감소시킨 것을 확인할 수 있다. 또한, 다수의

GPU 작업이 동시에 실행되면 GPU를 공유하는 컨테이너의 개수보다 각 컨테이너에서 실행하는 GPU 작업의 규모가 성능에 더 큰 영향을 주며, 본 논문에서 제안한 컨테이너 관리 기법은 GPU 자원 경쟁으로 인한 성능 저하를 감소시키고 GPU 작업의 규모가 큰 경우에 더 효과적으로 작동한다는 것을 확인할 수 있다. 또한, 본 논문에서 GPU 작업을 지연처리하기 위해 사용한 컨테이너 일시 정지 기능인 *docker pause*와 *unpause*는 약 0.06초의 실행 시간이 발생하기 때문에 GPU 작업의 성능에 미치는 영향은 매우 작다.

6. 관련 연구

컨테이너 기반 클라우드 환경에서 GPU 작업과 자원을 관리하기 위한 다양한 연구가 제안되었다. 클라우드 환경에서 다수의 사용자는 컴퓨팅 자원을 공유하기 때문에 자원 활용률과 경쟁 사이의 균형을 조절하기 위한 기법이 필요하다. 클라우드 환경에서는 고비용 연산 장치인 GPU를 효율적으로 활용하고 다수의 사용자에게 공유하기 위한 다양한 기법들이 제안되었다.

적응적 GPU 공유 및 스케줄링 기법[11]은 컨테이너 환경에서 다수의 딥러닝 및 고성능 연산 작업이 GPU를 공유하는 환경에서 특정 컨테이너가 GPU 메모리를 대부분 점유할 때 GPU 메모리 부족으로 인해 다른 작업이 실행되지 못하는 문제를 해결하기 위해 GPU 메모리 할당 작업을 조정하여 컨테이너의 GPU 메모리 사용량을 제한하고 이를 통해 특정 컨테이너의 GPU 메모리 독점을 방지한다. 적응적 GPU 공유 및 스케줄링 기법은 컨테이너 환경에서 GPU 메모리를 각 컨테이너에 공정하게 분배하고 각 컨테이너의 실행 순서를 조절하여 메모리 부족으로 인한 작업 실행 실패를 방지한다. gShare[12]는 GPU 메모리 부족으로 인해 GPU 작업 실행이 불가능해지는 문제를 해결하기 위해 GPU 메모리 작업을 조절할 수 있도록 API Remoting 방식을 사용하며, GaiaGPU[13]는 GPU 자원 할당과 스케줄링을 위해 API 인터셉트를 사용해 자원 할당 작업을 관리하며, 짧은 작업을 위한 GPU 스케줄링 기법[14] 또한 GPU 자원을 할당하고 사용하는데 자원 관리를 위한 수정된 API를 활용해 자원 할당과 GPU 작업의 실행을 조절한다.

Kube-Knots[15]은 컨테이너 기반 클러스터 관리 시스템인 쿠버네티스[16] 환경에서 지연시간에 민감한 GPU 작업에 대해 정밀한 자원 할당이 제공되지 못하는 문제를 해결하기 위해 Correlation 기반 예측과 피크(Peak) 예측 기반 스케줄링 방식을 사용해 GPU 활용도를 개선한다. Kube Share[17] 또한, 쿠버네티스 환경에서 다수의 컨테이너가 단일 GPU를 공유하지 못하는 기존 환경의 문제를 해결하기 위해 다수의 컨테이너가 GPU를 공유할 수 있도록 지원한다. Optimus[18]은 컨테이너 기반 클러스터 환경에서 딥러닝 작업을 효율적으로 수행하기 위해 딥러닝 모델을 활용하여 할당된 자원을 통한 학습 속도를 추정하여 자원을 동적으로 할당하는 방법을 사용한다.

KubFBS[19]는 다수의 딥러닝 작업이 동시에 실행될 때 자원 경쟁을 해결하기 위해 자원 사용 특성을 고려한 fine-grained and balance-aware scheduling(FBSM) 기법을 제안하고 노드의 GPU 메모리 사용량을 기반으로 클러스터의 GPU 자원 부하분산을 통해 딥러닝 작업의 실행 속도를 가속화한다. Gemini[20]는 GPU 할당 작업을 지원하기 위해 CUDA의 kernel 함수 실행과 메모리 할당과 관련된 API의 후킹을 통한 동적 시분할 스케줄링이 가능한 user-space 런타임 스케줄링 프레임워크를 제안한다. 그리고 AntMan[21]은 다수의 딥러닝 작업이 GPU를 공유하고 동시에 실행되는 환경에서 딥러닝 작업 사이의 간섭을 방지하기 위해 딥러닝 프레임워크인 Tensorflow[22]와 PyTorch[23] 내부에 메모리와 계산 자원의 동적 스케일링 메커니즘을 구현했다.

기존 연구는 GPU 메모리 부족으로 인한 작업 실패에 대해 초점을 두고 있으며, 다수의 GPU 작업이 GPU를 공유하는 환경에서 GPU 코어 경쟁에 대한 문제를 고려하고 있지 않다. 또한, 몇몇 기존 연구는 수정된 GPGPU API나 라이브러리가 필요하며, 이는 컨테이너 사용자가 클라우드 관리자에 의해 제공된 API 기능 및 버전만 사용해야 하기 때문에 GPU 애플리케이션 구현 시 투명성을 완전하게 제공하지 못한다.

본 논문에서 제안한 컨테이너 관리기법은 컨테이너 이미지, GPU 작업의 소스 코드와 같이 컨테이너 사용자 측면의 어떠한 구성 요소도 수정할 필요가 없으며, 컨테이너 사용자가 프로그램 개발 시 GPU 자원 관리를 위한 기능을 컨테이너 사용자가 고려할 필요 없다. 또한, 자원 사용량 및 GPU 작업 실행 상태에 대한 정보를 클라우드 관리자 측면에서 얻을 수 있는 모니터링 정보만으로 추적하기 때문에 작업 규모, 자원 요구량과 같이 GPU 작업이나 사용자가 제공해야 알 수 있는 어떠한 정보도 필요로 하지 않기 때문에 완전하게 클라우드 관리자 측면에서 얻을 수 있는 정보만으로 컨테이너를 관리할 수 있다.

7. 결론 및 향후 연구

본 논문에서는 GPU가 공유된 컨테이너 기반 클라우드 환경에서 다수의 컨테이너가 GPU 작업을 동시에 실행할 때 GPU 자원 경쟁으로 인한 성능 저하를 방지하기 위한 컨테이너 관리기법을 제안한다. 이전 장의 실험 결과와 같이 기존 컨테이너 환경에서 다수의 컨테이너가 GPU를 공유하여 GPU 작업을 수행할 때 다수의 GPU 프로세스가 단일 GPU에서 실행될 때 발생하는 컨텍스트 전환은 GPU 작업의 실행 시간에 오버헤드를 발생시켰으며, 컨테이너에서 실행되는 GPU 작업의 성능이 클수록 오버헤드는 증가했다.

본 논문에서 제안한 컨테이너 관리기법은 컨테이너의 생명주기를 관리 기능을 활용하여 GPU 작업을 수행하는 컨테이너

가 존재할 때 GPU 자원 사용량을 기반으로 컨테이너를 일시 정지시켜 GPU 작업을 지연시켜 GPU 자원 경쟁을 방지한다. 제안한 기법은 GPU 작업의 지연 처리를 통해 GPU 자원 경쟁을 회피하기 때문에 이전 장의 실험에서 설명한 것과 같이 GPU 자원 경쟁으로 인한 성능 저하를 완화하여 컨테이너에서 실행되는 GPU 작업의 전체 성능을 개선하였다. 실험 결과에서 설명한 것과 같이 컨테이너에서 실행하는 GPU 작업의 규모가 클수록 효율적으로 작동하였으며, GPU 자원 경쟁 상대가 없는 경우에 전체 성능이 향상된 것을 확인할 수 있다.

본 논문에서는 GPU 작업을 지연 처리하기 위해 GPU 메모리 입력 작업 시 컨테이너의 GPU 메모리 사용량이 0 이상일 때 컨테이너를 일시 정지시켰다. 이는 컨테이너에서 실행되는 GPU 작업의 GPU 메모리 사용량을 알 수 없어서 일시 정지할 수 없는 GPU 코어를 사용한 연산 작업이 시작되기 전에 GPU 메모리 사용량이 0 이상이면 GPU 작업이 실행됐다고 판단하고 컨테이너를 일시 정지하기 위함이다. 하지만 다른 컨테이너의 GPU 작업이 실행 중인 사이에 컨테이너의 GPU 메모리 입력 작업이 완료되고 컨테이너를 일시 정지하면 다음 실행되는 컨테이너의 GPU 작업은 GPU 메모리 입력 작업이 완료되었기 때문에 GPU 코어를 사용한 연산 작업을 즉시 실행하고 GPU 작업을 실행하는 컨테이너들의 전체 성능을 최적화할 수 있을 것이다.

본 논문에서 GPU 작업의 완료를 확인하기 위해 GPU 메모리 사용량을 참조하는 방법은 완전하게 외부에서 GPU 작업의 실행 상태를 추적하면서 GPU 프로세스가 실제 종료되기 전에 GPU 공유 대상에서 제외할 수 있다. 하지만 GPU 프로세스를 유지하면서 kernel 함수를 여러 번 실행하거나 GPU 작업에 사용될 데이터의 크기가 GPU 메모리 용량을 초과하여 GPU 메모리에 데이터를 빈번하게 할당하고 해제하는 경우, GPU 작업이 실행되는 중간에 GPU 메모리 사용량이 0이 되어 GPU 공유 대상에서 제외된다. 이러한 경우 해당 GPU 작업이 다시 GPU 메모리를 사용하게 되면 본 논문에서 제안한 방식으로 GPU 사용 대상에 포함되어 GPU 작업을 실행할 것이다.

추후 연구에서 각 컨테이너의 GPU 메모리 사용량이나 작업 규모 그리고 패턴에 대한 SLA 및 모니터링 정보를 활용하여 컨테이너가 GPU 작업을 수행할 때 GPU 연산 작업의 대기시간을 감소시키기 위한 연구를 수행할 계획이다. 또한, 이전 장에서 수행한 실험에서 각 컨테이너에서 실행하는 GPU 작업의 규모가 작으면 본 논문에서 제안한 컨테이너 관리기법이 비효율적으로 작동하는 것을 확인하였으며, 각 컨테이너에서 실행되는 GPU 작업은 모두 동일한 GPU 작업을 사용했다. 하지만 실제 클라우드 환경에서 각 컨테이너에서 실행될 작업의 규모는 다양할 것이며, 자원 사용량 및 실행 시간이 서로 상이할 것이다. 이러한 특성을 고려하여 GPU 작업의 규모와 GPU를 공유하는 컨테이너의 개수에 대한 정보를 컨테이너 관리를 위한 메트릭으로 활용하여 본 논문에서 제안한 기법을 개선할 계획이다.

References

[1] NVIDIA, NVIDIA Docker Wiki [Internet], https:// github.com/NVIDIA/nvidia-docker/wiki.

[2] Docker, Docker [Internet], https://www.docker.com/.

[3] Docker, Docker CLI [Internet], https://docs.docker.com/engine/reference/commandline/pause/.

[4] NVIDIA, Compute Unified Device Architecture (CUDA) [Internet], https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html.

[5] NVIDIA, CUDA C++ Programming Guide [Internet], https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[6] NVIDIA, Multi-Process Service(MPS) [Internet], https://docs.nvidia.com/deploy/mps/index.html.

[7] NVIDIA, NVIDIA Docker [Internet], https://github.com/NVIDIA/nvidia-docker.

[8] Docker, docker ps [Internet], https://docs.docker.com/engine/reference/commandline/ps/.

[9] Docker, docker top [Internet], https://docs.docker.com/engine /reference/commandline/top/.

[10] NVIDIA, NVIDIA System Management Interface [Internet], https://developer.nvidia.com/nvidia-system-management-interface.

[11] Q. Chen, J. Oh, S. Kim, and Y. Kim, "Design of an adaptive GPU sharing and scheduling scheme in container-based cluster," *Cluster Computing*, Vol.23, No.3, pp.2179-2191, 2020.

[12] M. Lee, H. Ahn, C. H. Hong, and D. S. Nikolopoulos, "gShare: A centralized GPU memory management framework to enable GPU memory sharing for containers," *Future Generation Computer Systems*, Vol.130, pp.181-192, 2022.

[13] J. Gu, S. Song, Y. Li, and H. Luo, "GaiaGPU: sharing GPUs in container clouds," In *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, pp.469-476, 2018.

[14] J. Shao, J. Ma, Y. Li, B. An, and D. Cao, "GPU scheduling for short tasks in private cloud," In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pp.215-2155, 2019.

[15] P. Thinakaran, J. R. Gunasekaran, B. Sharma, M. T. Kandemir, and C. R. Das, "Kube-knots: Resource harvesting through dynamic container orchestration in gpu-based datacenters," In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pp.1-13, 2019.

[16] Linux Foundation, kubernetes [Internet], https://kubernetes.io/ko/.

[17] T. A. Yeh, H. H. Chen, and J. Chou, "Kubeshare: A framework to manage gpus as first-class and shared resources in container cloud," In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, pp.173-184, 2020.

[18] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," In *Proceedings of the Thirteenth EuroSys Conference*, pp.1-14, 2018.

[19] Z. Liu, C. Chen, J. Li, Y. Cheng, Y. Kou, and D. Zhang, "KubFBS: A fine-grained and balance-aware scheduling system for deep learning tasks based on kubernetes," *Concurrency and Computation: Practice and Experience*, Vol.34, No.11, pp.e6836, 2022.

[20] H. H. Chen, E. T. Lin, Y. M. Chou, and J. Chou, "Gemini: Enabling multi-tenant gpu sharing based on kernel burst estimation," *IEEE Transactions on Cloud Computing(Early Access)*, 2021.

[21] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia, "AntMan: Dynamic scaling on GPU clusters for deep learning," In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pp.533-548, 2020.

[22] Google Brain, Tensorflow [Internet], https://www.tensorflow.org/.

[23] Facebook AI Research, PyTorch [Internet] https://pytorch.org/.

강 지 훈

https://orcid.org/0000-0003-4773-6157
e-mail : k2j23h@korea.ac.kr
2011년 배재대학교 게임공학과(학사)
2013년 배재대학교 게임멀티미디어공학과 (석사)
2020년 고려대학교 컴퓨터학과(박사)
2020년 ~ 현 재 고려대학교 4단계 BK21 컴퓨터학교육연구단 연구교수
관심분야 : Cloud Computing, GPU Virtualization, HPC Cloud