KIPS Trans. Comp. and Comm. Sys. 클러스터 기반 컨테이너 환경에서 실시간 GPU 작업 모니터링 및 컨테이너 배치를 위한 노드 리스트 관리기법 381
Vol.11, No.11 pp.381~394 https://doi.org/10.3745/KTCCS.2022.11.11.381
ISSN: 2287-5891 (Print), ISSN 2734-049X (Online)

Real-Time GPU Task Monitoring and Node List Management Techniques for Container Deployment in a Cluster-Based Container Environment

Jihun Kang† · Joon-Min Gil††

ABSTRACT

Recently, due to the personalization and customization of data, Internet-based services have increased requirements for real-time processing, such as real-time AI inference and data analysis, which must be handled immediately according to the user's situation or requirement. Real-time tasks have a set deadline from the start of each task to the return of the results, and the guarantee of the deadline is directly linked to the quality of the services. However, traditional container systems are limited in operating real-time tasks because they do not provide the ability to allocate and manage deadlines for tasks executed in containers. In addition, tasks such as AI inference and data analysis basically utilize graphical processing units (GPU), which typically have performance impacts on each other because performance isolation is not provided between containers. And the resource usage of the node alone cannot determine the deadline guarantee rate of each container or whether to deploy a new real-time container. In this paper, we propose a monitoring technique for tracking and managing the execution status of deadlines and real-time GPU tasks in containers to support real-time processing of GPU tasks running on containers, and a node list management technique for container placement on appropriate nodes to ensure deadlines. Furthermore, we demonstrate from experiments that the proposed technique has a very small impact on the system.

Keywords : HPC Cloud, Container, GPU Computing, Real-time Task, Monitoring

클러스터 기반 컨테이너 환경에서 실시간 GPU 작업 모니터링 및 컨테이너 배치를 위한 노드 리스트 관리기법

강 지 훈† · 길 준 민††

요 약

최근 인터넷 기반 서비스는 데이터의 개인화 및 맞춤화로 인해 사용자의 상황이나 요구사항에 따라 즉시 처리해야 하는 실시간 AI 추론 및 데이터 분석과 같은 실시간 처리에 대한 요구사항이 증가하고 있다. 실시간 작업은 각 작업이 시작되고 결과를 반환하기까지의 데드라인이 정해져 있으며, 데드라인의 보장은 서비스의 품질과 직접적으로 연결된다. 하지만, 기존 컨테이너 시스템에서는 컨테이너에서 실행되는 작업의 데드라인을 할당하고 관리하기 위한 기능이 제공되지 않기 때문에 실시간 작업을 운용하는데 제한적이다. 또한, AI 추론 및 데이터 분석과 같은 작업은 GPU(Graphic Processing Unit)를 기본적으로 사용하는데, 일반적으로 GPU 자원은 컨테이너 사이에 성능 격리가 제공되지 않기 때문에 서로 성능 영향을 미치며, 노드의 자원 사용량만으로는 각 컨테이너의 데드라인 보장률이나 새로운 실시간 컨테이너의 배치 여부를 결정할 수 없다. 따라서, 본 논문에서는 컨테이너에서 실행되는 GPU 작업의 실시간 처리를 지원하기 위해 컨테이너의 데드라인 및 실시간 GPU 작업의 실행 상태를 추적하고 관리하기 위한 모니터링 기법과 클러스터 환경에서 실시간 GPU 작업을 실행하는 컨테이너가 데드라인을 보장할 수 있도록 적절한 노드에 배치하기 위한 노드 리스트 관리기법을 제안한다. 또한, 실험을 통해 제안하는 기법이 시스템에 매우 작은 영향을 미친다는 것을 증명한다.

키워드 : 고성능 클라우드, 컨테이너, GPU 컴퓨팅, 실시간 작업, 모니터링

1. 서 론

GPU를 활용한 고성능 클라우드 환경은 클라우드 인프라 사용자에게 GPU 장치를 제공하여 AI 학습, 빅데이터 처리와 같이 고성능 연산이 필요한 작업을 고속으로 처리할 수 있도록 지원한다. 최근 데이터의 개인화 및 사용자의 상황이나 요구에 따라 맞춤화가 됨에 따라 미리 처리된 데이터를 출력하는 것이 아닌 서비스 요청 즉시 필요한 데이터를 처리

※ 이 논문은 2022년도 정부(교육부)의 재원(2022R1I1A1A01063551)과 2019년도 정부(과학기술정보통신부)의 재원(NRF-2019R1F1A1062039)으로 한국연구재단의 지원을 받아 수행된 연구임.
※ 이 논문은 2022년 한국정보처리학회 ASK 2022에서 "컨테이너 기반 클라우드 환경에서 실시간 GPU 작업을 지원하기 위한 데드라인 정보 관리 기법[1]"의 제목으로 발표된 논문을 확장한 것임.
† 정 회 원 : 고려대학교 4단계 BK21 컴퓨터학교육연구단 연구교수
†† 종신회원 : 대구가톨릭대학교 컴퓨터소프트웨어학부 교수
Manuscript Received : August 1, 2022
Accepted : August 12, 2022
* Corresponding Author : Joon-Min Gil(jmgil@cu.ac.kr)

해야 하는 실시간 AI 추론이나 데이터 분석과 같은 실시간 작업 처리의 요구사항이 증가하고 있다.

실시간 작업은 미리 정해진 시간 내에 사용자의 요청을 처리하고 결과를 반환해야 한다. 이에 따라 실시간 작업에는 작업의 마감 시간인 데드라인(deadline)이 설정되어 있으며, 실시간 서비스 제공자는 실시간 서비스의 데드라인 보장을 목표로 서비스를 운용한다. 실시간 서비스를 운용하기 위해서는 각 작업의 데드라인 정보를 관리해야 하며, 데드라인의 경우 실시간 작업이 시작되고 계산되기 때문에 실시간 작업의 실행 상태를 항상 추적해야 한다. 또한, 다수의 사용자가 컴퓨팅 자원을 공유하는 클라우드 환경에서 다른 작업으로 인한 성능 영향은 작업의 성능 저하를 발생시키며, 이로 인해 데드라인을 지키지 못할 확률이 증가한다. 따라서, 자원 공유로 인한 성능 영향의 최소화는 각 작업이 균일한 성능을 달성하고 실시간 작업이 데드라인 이내에 처리될 확률을 증가시킬 수 있다.

하지만, 기존 오픈소스 기반 컨테이너 환경은 실시간 작업을 위해 컨테이너에 데드라인을 할당하고 컨테이너에서 실행되는 작업의 실행 상태를 추적하고 관리하기 위한 기능을 제공하지 않기 때문에 컨테이너 기반 클라우드 환경에서 실시간 작업을 운용하는데 제약이 있다. 또한, 일반적으로 클라우드 환경에서 각 사용자가 컴퓨팅 자원을 사용할 때 자원 사용의 공평성에 초점을 두기 때문에 데드라인과 같이 컨테이너의 작업 실행 시점에 따라 동적으로 변하는 우선순위를 자원 사용 정책에 적용하는데 제약이 따른다. 특히, GPU (Graphic Processing Unit)의 경우 일반적으로 다수의 사용자를 대상으로 자원 격리를 제공하지 않으며, GPU 내부의 스케줄러에 의해 자원 사용 순서가 결정되기 때문에 GPU 작업이 시작된 후에는 작업의 실행 순서를 조절할 수 없다. 이로 인해 다수의 사용자가 GPU를 공유하는 경우 성능 영향이 발생하며, 성능 영향으로 인한 성능 저하 문제는 실시간 GPU 작업의 데드라인 보장률이 저하되는 문제로 이어진다. 단일 실시간 GPU 작업이 GPU를 독점하면 성능 영향으로 인한 성능 저하를 방지할 수 있지만, 자원 활용률 저하로 인해 컴퓨팅 자원의 낭비를 발생시킨다.

본 논문에서는 컨테이너 기반 클라우드 환경에서 실시간 GPU 작업을 운용하기 위한 데드라인 정보 관리 및 실시간 GPU 작업의 실행 상태 추적을 위한 모니터링 기법과 실시간 GPU 작업을 실행하는 컨테이너 배치를 위한 노드 리스트 관리기법을 제안한다. 논문의 시스템 환경은 데드라인이 할당된 다수의 컨테이너가 단일 GPU를 공유하는 환경을 대상으로 하며, 할당된 데드라인은 Docker 엔진[2]에 의해 관리된다. 또한, 각 컨테이너에서는 GPU 작업을 위해 GPGPU (General-Purpose computing on Graphics Processing Units) 프로그래밍 모델인 CUDA(Compute Unified Device Architecture)[3]로 구현된 작업을 실행한다. 실시간 GPU 작업의 스케줄링과 실제 컨테이너 배치 영역은 본 논문의 추후 연구로 계획하고 있으며, 본 논문에서는 실시간 GPU 작업을 지원하기 위한 GPU 작업의 실행 상태 모니터링과 실시간 GPU 작업을 실행하는 컨테이너를 노드에 배치하기 위한 노드 리스트 관리기법에 초점을 둔다. 본 논문에서 제안하는 기법의 주요 기여는 다음과 같다.

- 본 논문에서는 컨테이너 기반 클라우드 환경에서 실시간 GPU 작업을 지원하기 위한 데드라인 정보 관리 및 실시간 GPU 작업 실행 상태 추적 모니터링 기법을 제안한다.
- 또한, 클러스터 기반 컨테이너 환경에서 노드에서 실행되고 있는 실시간 GPU 작업의 데드라인 정보를 기반으로 새로 생성된 컨테이너를 배치하기 위한 노드 리스트 관리 기법을 제안한다.
- 본 논문에서 제안하는 기법은 실시간 GPU 작업의 실행 상태를 추적하기 위해 GPU 애플리케이션의 소스 코드를 수정하거나 컨테이너에 별도의 모듈을 실행할 필요 없으며, 컨테이너와 비 종속적으로 실행된다.
- 실험을 통해 실시간 GPU 작업의 상태 추적과 컨테이너 배치를 위한 노드 리스트 관리기법을 위한 모니터링 작업이 전체 시스템에 미치는 성능 영향이 매우 작다는 것을 검증한다.

본 논문의 구성은 다음과 같다. 2장에서는 본 논문에서 제안하는 기법을 설명하기 위한 기반 기술을 설명하며, 3장에서는 실험을 통해 다수의 GPU 작업이 단일 GPU에서 동시에 실행될 때 발생하는 성능 영향 문제를 분석한다. 4장에서는 본 논문에서 제안하는 GPU 작업 실행 상태 추적 모니터링 기법과 컨테이너 배치를 위한 노드 리스트 관리기법에 관해 설명하고, 5장에서는 실험을 통해 본 논문에서 제안한 기법이 전체 시스템에 미치는 영향을 분석한다. 6장에서는 관련 연구를 설명하고 마지막 7장에서는 결론 및 향후 연구에 대해 설명한다.

2. 기반기술

2.1 컨테이너 환경에서의 GPU 공유

컨테이너 기반 클라우드 환경은 NVIDIA Docker[4]를 통해 컨테이너가 GPU를 사용할 수 있도록 지원한다. GPU를 사용할 수 있는 다수의 컨테이너는 단일 GPU에 동시에 접근할 수 있으며, 동시에 실행된 GPU 작업은 다중 프로세스 환경으로 실행되고 GPU 내부의 하드웨어 스케줄링 정책에 의해 GPU 자원을 공유하게 된다. Fig. 1은 본 논문에서 사용한 Docker에서의 GPU 사용 구조를 보여준다.

클라우드 환경에서 전통적인 컴퓨팅 자원인 CPU, 메모리, 그리고 스토리지의 경우 논리적인 다중화를 통해 자원을 격리하여 각 사용자에게 컴퓨팅 자원 일부를 독점할 수 있도록 할당할 수 있으며, 이를 위한 하드웨어 기술이 제공된다. 하지만 GPU는 특정 제품을 제외하고는 일반적으로 자원 격리

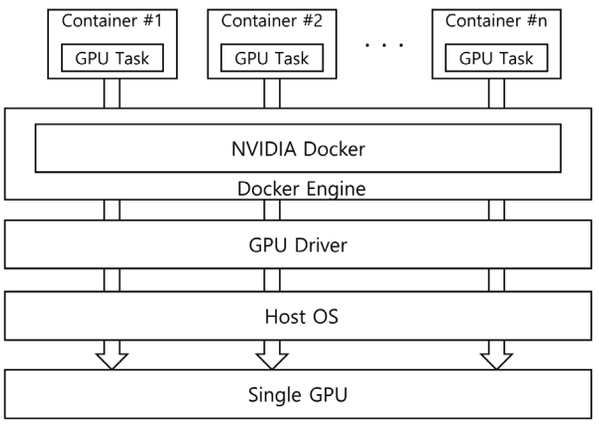

Fig. 1. GPU Sharing in Docker

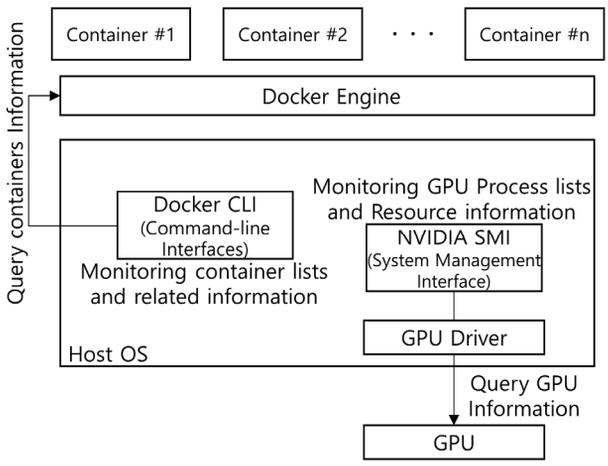

Fig. 2. GPU and Container Monitoring System

가 지원되지 않으며, 컨테이너 환경에서는 컨테이너의 GPU 사용량을 제한할 수 없기 때문에 각 사용자에게 GPU 자원의 일부를 독점적으로 할당할 수 없다.

다중 프로세스 환경에서 자원 격리 없이 자원을 공유하면 각 작업은 컴퓨팅 자원의 모든 영역에 접근할 수 있으며, 이로 인해 작업 사이에 자원 경쟁이 발생하고 상호 성능 영향을 끼치게 된다. 자원 경쟁으로 인한 성능 영향은 작업의 실행 시간에 대한 일관성을 보장하는 데 방해 요소가 된다. GPU를 공유하는 컨테이너가 증가할수록 각 작업의 실행 시간이 증가하며, 이는 실시간 GPU 작업의 데드라인 보장을 더 어렵게 만든다. 데드라인이 정해진 실시간 작업은 작업 완료 시간이 매우 중요하다. 하지만 앞서 설명한 것과 같이 일반적으로 GPU는 자원 격리를 제공하지 않기 때문에 성능 일관성을 제공하는데 제한적이며, 다수의 실시간 GPU 작업이 데드라인을 보장하면서 잘 운용되는 환경에서도 새로 생성된 컨테이너로 인해 실행 중인 실시간 GPU 작업의 성능 영향을 줄 수 있기 때문에 GPU를 공유하는 다수의 실시간 GPU 작업을 운용하기 위해서는 동시에 실행되는 다른 작업을 고려한 컨테이너 관리가 필요하다.

2.2 컨테이너 및 GPU 모니터링

클라우드 환경에서 사용자의 컨테이너를 관리할 때 자원 사용이나 실행 중인 컨테이너의 개수와 같은 모니터링 정보가 매우 중요하다. 클라우드 관리자는 이러한 자원 사용량과 관련된 모니터링 정보를 기반으로 각 사용자의 컨테이너를 관리하며, 자원 경쟁으로 인한 성능 영향이나 특정 노드의 부하 집중과 같은 관리 측면의 다양한 문제들이 발생하는 것을 방지하고 관리한다.

컨테이너 시스템인 Docker는 모니터링을 위한 다양한 명령어를 제공한다. *docker ps*[5]와 *docker stats*[6]와 같은 명령어는 서버에서 실행 중인 모든 컨테이너의 정보와 자원 사용량을 모니터링할 수 있으며 *docker top*[7]은 컨테이너에서 실행 중인 프로세스를 모니터링할 수 있다. 또한, 컨테이너 기반 클러스터 환경을 위한 Prometheus[8]의 경우 클러스터링 된 모든 노드의 자원 사용 정보를 cAdvisor[9]를 활용해 모니터링 메트릭을 수집한다. 하지만, GPU의 경우 컨테이너 시스템인 Docker가 아닌 GPU 관리 도구 NVIDIA Docker[10]에서 관리하며, GPU 자원 사용량, 실행 중인 프로세스와 같이 GPU 자원과 관련된 정보는 NVIDIA SMI (System Management Interface)[11]를 통해 모니터링된다. Fig. 2는 GPU 사용이 가능한 컨테이너 환경에서 컨테이너와 GPU의 모니터링 시스템의 구조를 보여준다.

일반적인 컨테이너 환경에서 컨테이너와 관련된 모니터링 정보와 GPU 자원과 관련된 모니터링 정보를 관리하는 모니터링 시스템은 독립적으로 작동한다. 이로 인해 컨테이너 모니터링 시스템에서는 GPU 작업에 대한 정보를 얻을 수 없고 GPU 모니터링 시스템에서는 컨테이너에 대한 정보를 얻을 수 없다. 또한, 노드의 자원 사용량에 대한 정보 수집에 초점을 두기 때문에 각 컨테이너의 작업 실행 상태는 알 수 없으며, 이러한 특성으로 인해 실시간 작업의 데드라인 계산이나 작업 실행과 데드라인 계산 작업을 수행하는 것이 불가능하다. 따라서 컨테이너 기반 클라우드 환경에서 실시간 GPU 작업을 운용하기 위해서는 컨테이너의 데드라인, GPU 작업의 실행 상태와 같이 실시간 작업을 관리하기 위한 정보를 추적하고 컨테이너와 GPU의 모니터링 정보를 통합할 필요가 있다.

2.3 클러스터 환경에서 컨테이너 프로비저닝(Provisioning)

클라우드 환경에서 사용자의 새로운 컨테이너의 생성 요청이 발생하면 가장 먼저 새로 생성된 컨테이너를 어떤 노드에 배치할지 결정해야 한다. 일반적으로 클러스터 기반 클라우드 환경에서 새로운 인스턴스를 생성하고 관리하는 작업은 노드의 자원 사용률을 기반으로 작동한다. 일반적인 클라우드 관리 시스템의 목적은 자원의 활용률을 높이면서 클러스터를 구성하는 노드의 부하를 분산하는 것이다. 이를 통해 자

원의 유휴시간을 최소화하고 특정 노드에 사용자가 집중되는 것을 방지해 부하 증가로 인한 성능 저하 문제를 해결한다.

클라우드 관리 시스템은 새로운 컨테이너를 배치하기 위해 클러스터를 구성하는 모든 노드의 자원 사용에 대한 모니터링 정보를 통합 관리하며, 앞서 설명한 것과 같이 다수의 노드를 클러스터 형태로 관리할 때 각 노드의 자원 사용률에 초점을 둔다. 하지만, 실시간 서비스를 운영할 때 가장 중요한 정보는 데드라인과 관련된 정보이다. 또한, 컨테이너에 할당된 데드라인을 기반으로 각 컨테이너의 남은 데드라인 시간을 계산하기 위해서는 작업의 실행과 종료와 같이 작업의 실행 상태에 대한 정보도 매우 중요하다.

일반적인 클라우드 환경에서는 클러스터의 노드를 관리하는 데 가장 중요한 성능 메트릭은 자원 사용량이다. 클라우드 관리자는 자원 사용량 정보를 기반으로 사용자가 요구하는 자원 용량에 맞는 노드를 선택하여 배치함으로써 특정 노드에 사용자가 집중되는 것을 방지하고 컴퓨팅 자원이 초과 사용되어 성능이 저하되는 문제를 해결한다. 하지만, 실시간 서비스를 운용할 때 가장 중요한 성능 메트릭은 데드라인이다. 클라우드 관리자는 서버에서 실행 중인 모든 실시간 작업의 데드라인을 보장하기 위해 컴퓨팅 자원의 부하분산보다는 데드라인이 짧은 작업이 특정 노드에 집중되지 않도록 분산시켜야 하며, 새로운 컨테이너를 배치할 때 자원 사용량보다는 데드라인 경쟁이 발생하지 않도록 하는 것이 더 중요하다.

특히, 실시간 GPU 작업을 운용하는 환경에서는 앞서 설명한 것과 같이 GPU 장치는 일반적으로 자원 격리를 제공하지 않기 때문에 다수의 컨테이너가 GPU를 공유할 때 GPU를 공유하는 모든 컨테이너가 성능 영향을 받는다. 이로 인해, 새로 실행된 컨테이너로 인한 자원 초과 사용은 새로 실행된 컨테이너뿐만 아니라 기존에 실행되고 있던 다른 컨테이너의 성능에도 영향을 미치기 때문에 특정 시간 내에 작업을 완료해야 하는 실시간 작업에 악영향을 끼친다. 또한, 가용 자원이 충분하더라도 데드라인이 짧은 작업이 특정 노드에 집중되게 된다면 데드라인 경쟁으로 인해 데드라인을 보장하지 못하는 경우도 발생할 수 있다.

이러한 특성으로 인해 실시간 GPU 작업을 운용하는 클라우드 환경에서는 데드라인 보장에 방해되는 요소를 최소화하기 위해 각 컨테이너의 데드라인, 실시간 GPU 작업의 실행 상태와 같은 실시간 작업과 관련된 정보를 추적하고 수집된 정보를 기반으로 새로운 컨테이너를 노드에 배치할 때 기존에 실행되고 있던 실시간 GPU 작업을 방해하지 않도록 적절한 노드에 배치하기 위한 노드 선택 방법이 필요하다. 즉, 기존 컨테이너 기반 클라우드 환경과 같이 자원 사용량을 기반으로 하는 컨테이너 관리 정책이 아닌 노드에서 최대한 많은 실시간 GPU 작업을 운용하면서 실시간 GPU 작업의 데드라인 정보를 기반으로 하는 컨테이너 관리 기법이 필요하며, 이를 위한 데드라인 정보 관리 및 실시간 GPU 작업의 실행 상태 추적이 필요하다.

본 논문에서는 앞서 설명한 것과 같이 기존 컨테이너 환경에서 실시간 GPU 작업을 운용하는데 발생하는 제한점을 해결하기 위해 오픈소스 기반 컨테이너 시스템인 Docker를 수정하여 컨테이너 설정 파일에 데드라인 정보를 입력하고 저장하기 위한 데드라인 관리 기능을 추가하고 실시간 GPU 작업의 남은 데드라인을 계산하기 위해 GPU 작업의 시작과 종료 시점을 찾기 위한 GPU 작업 실행 상태 모니터링 기법을 제안한다. 또한, 새로 생성된 컨테이너로 인해 노드에서 실행중인 컨테이너 사이에서 발생하는 데드라인 경쟁을 방지하기 위한 노드 리스트 관리기법을 제안한다.

3. 관련 연구

데드라인 제약과 데이터 지역성을 인지한 스케줄링 기법[12]은 멀티테넌시 환경에서 서비스의 QoS를 보장하고 자원 활용률을 향상하기 위해 스케줄링 정책을 동적으로 관리하는 기법을 제안한다. 비용 효율적 스케줄링 기법[13]은 모바일 엣지 컴퓨팅 환경에서 지연 시간에 민감한 실시간 작업의 비용 최적화를 제안하며, 이를 반영하여 엣지 서버의 작업을 스케줄링하고 데드라인을 보장하기 위해 자원 성능이 높은 클라우드 자원을 활용한다. 데드라인을 고려한 동적 자원 관리 기법[14]은 데드라인에 민감한 작업의 리소스를 동적으로 관리하여 작업의 응답 시간을 개선한다. 해당 방법은 서버리스 환경에서 함수를 배치할 때 동적 자원 관리와 부하 분산을 통해 데드라인을 보장할 수 있도록 지원한다.

앞서 설명한 기존 연구는 실시간 작업의 데드라인 보장을 위해 자원의 동적 관리를 통해 데드라인 위반을 방지한다. 하지만, 데드라인 경쟁으로 인한 데드라인 위반은 특정 작업에 자원을 추가 할당하더라도 데드라인을 보장할 수 없다. 데드라인 경쟁은 클러스터 차원의 모니터링 정보 통합 관리를 통한 데드라인 측면의 부하 분산이 필요하다.

REACT[15]는 컨테이너 오케스트레이션 시스템인 쿠버네티스를 기반으로 실시간 컨테이너를 위한 오케스트레이터 아키텍처를 제안한다. 해당 방법은 실시간 컨테이너의 성능 측정을 위한 메트릭을 정의하고 실시간 컨테이너와 비실시간 컨테이너를 노드에 혼합 배치하여 실시간 컨테이너의 데드라인을 보장한다. IoT 작업 컨테이너화 기술[16]은 산업 IoT 환경에서 허용 응답 시간 내에 특정 작업을 실행할 때 필요한 전용 장비에 대한 종속성을 해결하기 위해 호스트 시스템의 가용 자원을 통해 애플리케이션을 컨테이너에서 실행하는 기법을 제안한다. 해당 방법은 게임 이론 방식을 통해 컨테이너화될 작업의 비율을 추정하고 시스템 활용률을 최대화한다. SHWS[17]는 오프라인 배치 워크플로우와 온라인 스트림 워크플로우를 스케줄링하고 작업의 보조 데드라인과 우선순위를 지정하여 작업의 데드라인과 처리 종속성 요구사항을 보장하기 위한 스케줄링 기법을 제안한다. 다목적 CR-PSO 스

케줄링 기법[18]은 데드라인이 할당된 여러개의 클라우드렛의 데드라인을 보장하기 위한 스케줄링을 위해 부분 군집 최적화(Partial Swarm Optimization)와 화학 반응 최적화(Chemical Reaction Optimization) 알고리즘을 결합한 하이브리드 스케줄링 알고리즘을 제안한다.

클라우드 환경에서 데드라인이 할당된 실시간 작업 운용을 위한 기존 연구는 자원 관리 정책을 데드라인에 적응적으로 관리함으로써 실시간 작업이 데드라인 이내에 처리될 수 있도록 지원하기 위한 기술들을 제안한다. 하지만, 실행되고 있는 실시간 작업이 실제 데드라인 이내에 처리되는지 실시간 작업의 실행 상태를 추적하기 위한 기법이 제공되지 않으며, 작업의 실제 실행 시간을 고려하지 않고 자원 사용량과 데드라인 정보를 기반으로 컨테이너를 스케줄링하는 방법을 사용하기 때문에 다양한 실제 실행 시간에 비해 데드라인이 촉박한 작업에 대해서 대응하는데 제한적이다. 또한, 기존 연구는 실시간 작업의 실행 상태를 실시간으로 추적하지 않기 때문에 데드라인 위반 상황을 실시간으로 알 수 없어서 서비스 운용 중 QoS 위반 상황에 대처하는데 제한적이다. 하지만 본 논문에서 제안한 기법은 실시간 GPU 작업의 실행 상태를 실시간으로 추적하기 때문에 데드라인 위반 상황을 런타임에 알 수 있으며, 이를 통해 컨테이너의 이주와 같은 부하 분산 기법을 적용한다면 데드라인 위반 문제를 해소할 수 있다.

4. GPU를 공유하는 환경에서 자원 경쟁으로 인한 성능 저하

이전 장에서 설명한 것과 같이 일반적으로 GPU는 자원 격리 기술을 제공하지 않는다. 자원 격리 기술이 제공되지 않기 때문에 컨테이너가 사용할 수 있는 GPU 자원의 용량을 제한할 수 없으며, 이로 인해 GPU를 공유하는 컨테이너 환경에서 각 컨테이너는 GPU 작업에 따라 가용 자원을 최대한 많이 사용하도록 작동하게 된다. 이러한 특성은 GPU를 공유하여 실행되는 GPU 작업 사이에서 자원 경쟁을 발생시키며, 상호 성능 영향을 미쳐 GPU를 공유하는 대상이 증가할수록 실행 중인 전체 GPU 작업의 성능 저하 문제를 발생시킨다.

이번 장의 실험에서는 다수의 컨테이너가 GPU를 공유하는 환경에서 GPU 작업을 실행하는 컨테이너의 성능 측정을 통해 GPU 자원의 사용량 및 GPU를 공유하는 컨테이너의 개수에 따른 GPU 작업의 성능 저하 문제를 분석한다. 실험에서 사용할 GPU 작업은 GPGPU 프로그래밍 API인 CUDA로 구현된 행렬 곱셈 작업을 수행하며, GPU를 공유하는 컨테이너의 개수가 증가할 때의 평균 성능과 각 컨테이너 사이의 성능 편차를 측정한다. 실험에서는 GPU 코어 부하를 높이기 위해 20개의 컨테이너가 각각 CUDA로 구현한 8,192×8,192 행렬 곱셈을 수행한다. 실험 결과는 Fig. 3과 같다.

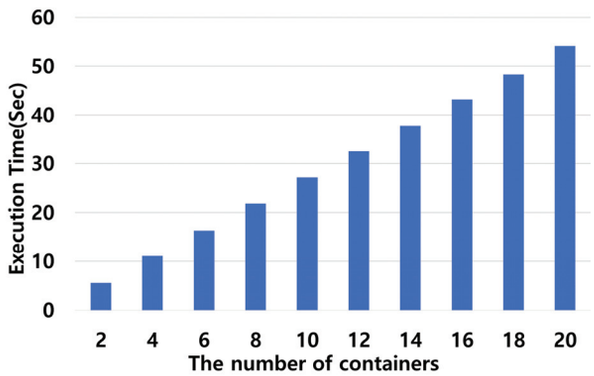

Fig. 3. Increased Task Run Time as GPU Core Loads Increase

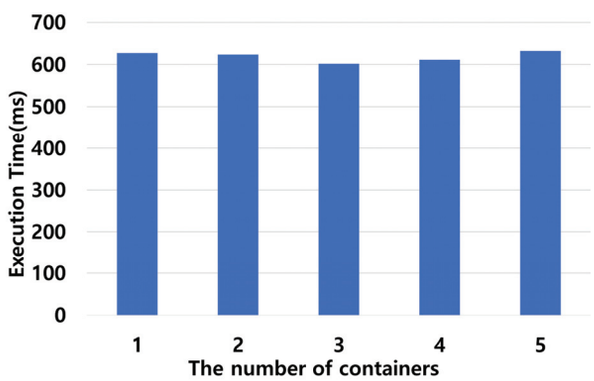

Fig. 4. Performance When GPU Core Usage is Less Than 100%

Fig. 3의 실험에서 컨테이너에서 실행되는 행렬 곱셈 작업은 67,108,864개의 스레드를 생성한다. 하나의 컨테이너가 GPU 코어를 100% 사용하게 되며, 행렬 곱셈을 수행하는 컨테이너가 증가할수록 GPU 코어의 부하가 증가하게 되고 GPU 코어가 한 번에 처리할 수 있는 스레드의 범위를 넘어서게 된다. 이로 인해 동시에 실행되는 컨테이너의 개수가 증가할수록 GPU 코어에서 처리되지 못하고 대기하는 스레드가 증가하게 되며, 이로 인해 전체적인 실행 시간이 증가하게 된다. 하지만 Fig. 4와 같이 GPU 코어의 사용량이 100% 이하일 경우 동시에 실행되는 컨테이너의 개수와 상관없이 균일한 성능을 얻게 된다. Fig. 4의 실험은 Tensorflow[19]를 사용한 MNIST[20] 데이터셋의 추론 작업의 성능을 보여준다. 컨테이너는 최대 5개를 사용하며, 미리 생성된 AI 모델을 사용해 10,000개의 이미지 추론 작업을 수행한다. 실험 결과에서는 모델 데이터의 입력시간은 제외하고 순수 추론 작업의 실행 시간을 보여준다.

Fig. 4의 실험 결과와 같이 컨테이너 1개에서 추론 작업을 수행했을 때와 컨테이너 5개에서 추론 작업을 동시에 실행했을 때의 성능 차이가 크게 발생하지 않는다. Fig. 4의 실험에서 사용한 추론 작업은 5개의 컨테이너에서 동시에 실행해도 GPU 코어의 약 70% 이하로 사용하기 때문에 GPU 코어의 허용 범위 내에서 처리된다. 이로 인해 동시에 실행되는 컨테

이너의 개수가 증가해도 1개의 컨테이너에서 추론 작업을 실행했을 때와 성능 차이가 거의 발생하지 않게 된다.

Fig. 3의 실험 결과에서 보여주는 것과 같이 GPU 작업을 실행하는 컨테이너의 개수가 증가할수록 이에 따라 GPU 자원의 부하가 증가하며, GPU 자원의 부하로 인해 모든 컨테이너의 GPU 작업 실행 시간이 증가하는 것을 볼 수 있다. 또한, Fig. 4의 실험 결과에서 보여주는 것과 같이 GPU 코어를 100% 이하로 사용한 경우 동시에 실행되는 컨테이너가 늘어나도 균일한 성능을 달성하는 것을 볼 수 있다.

앞서 설명한 것과 같이 컨테이너 환경에서 GPU 자원은 GPU를 공유하는 다수의 컨테이너 사이에 자원 격리를 제공하지 않는다. 자원 격리를 제공하지 않는다는 것은 각 컨테이너의 GPU 자원 사용량을 제한할 수 없다는 것이며, 이로 인해 GPU 작업의 규모에 따라 가용 GPU 자원 모두를 사용하려고 할 것이다. 따라서, 컨테이너 사이에 자원 사용으로 인한 경쟁이 발생하며, GPU를 공유하는 모든 컨테이너의 성능에 영향을 미친다.

자원을 공유하는 컨테이너 사이에서 성능 격리가 불가능하다는 것은 GPU 자원을 공유하는 컨테이너의 개수에 따라 성능이 다르다는 것을 뜻하며, 이는 데드라인이 할당된 실시간 작업을 운용하는 환경에서 QoS를 보장하는데 제약사항을 발생시킨다. 하지만, 컨테이너 사용자가 실행하는 GPU 작업의 규모와 자원 사용량은 GPU 작업이 실행되고 완료되기 전까지 미리 알 수 없으며, 자원 사용량에 대한 정보만 가지고 실시간 작업의 데드라인이 보장되고 잘 운용되는지 알 수 없다.

따라서, 본 논문에서는 기존 컨테이너 기반 클라우드 환경에서 실시간 GPU 작업을 운용할 때 발생하는 제약을 해결하기 위해 실시간 GPU 작업의 실행 상태를 추적하기 위한 모니터링 기법과 새로 실행되는 컨테이너를 적절한 노드에 배치할 때 참조하기 위한 노드 리스트 관리기법을 제안한다. 본 논문에서 제안하는 기법에 대한 자세한 내용은 다음 장에서 자세히 설명한다.

5. 구 현

본 논문에서는 다수의 컨테이너가 GPU를 공유하는 클라우드 환경에서 실시간 GPU 작업을 운용하기 위한 실시간 GPU 작업 실행 상태 추적 기법과 노드 리스트 관리기법을 제안한다. 실시간 GPU 작업의 실행 상태를 추적하기 위해 실시간 GPU 작업 모니터링 시스템과 새로운 컨테이너 배치를 위한 노드 리스트 관리 기법을 제안한다. Fig. 5는 본 논문에서 제안하는 전체 시스템 구조를 보여준다.

본 논문에서 제안한 기법은 크게 3가지 서브 시스템으로 구성된다. 첫 번째는 컨테이너를 생성할 때 컨테이너의 설정 파일에 데드라인 정보를 입력하고 관리할 수 있도록 Docker 엔진과 Docker CLI(Command Line Interface)[21] 사이에

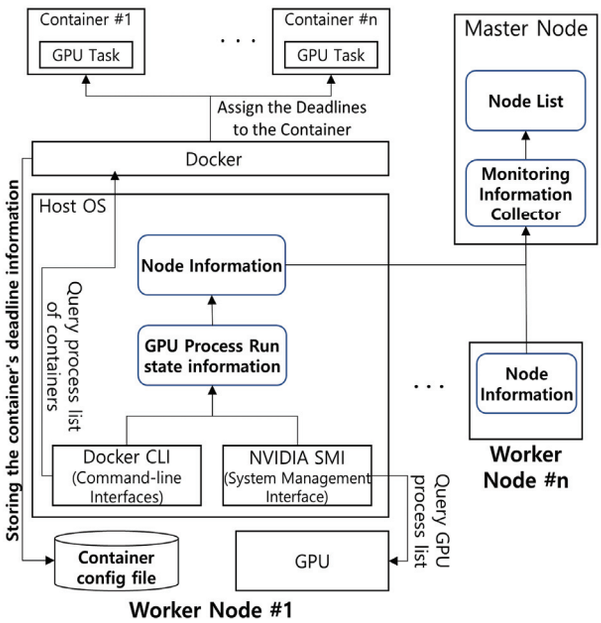

Fig. 5. Overall Structure of the System

데드라인 정보를 교환할 수 있는 데드라인 정보 관리 모듈. 두 번째는 GPU 자원 사용 상태를 기반으로 컨테이너의 GPU 작업 실행 상태를 추적하고 작업의 시작 및 종료 시점을 추출하기 위한 실시간 GPU 작업 실행 상태 모니터링 시스템, 마지막은 새로운 컨테이너가 배치될 때, 클러스터를 구성하는 각 노드의 데드라인 보장률 및 데드라인 여유시간 정보를 기반으로 컨테이너를 배치할 노드를 선택하기 위한 노드 리스트 관리 기법이다. 각 서브 시스템은 다음 절에서 자세히 기술한다.

5.1 실시간 GPU 작업 상태 모니터링

실시간 작업을 운용할 때 가장 중요한 정보는 데드라인 정보이다. 각 실시간 작업이 시작된 후 데드라인으로 정해진 특정 시간 이내에 작업을 완료해야 하기 때문에 데드라인 정보는 실시간 서비스를 운용하는 서비스 관리자의 요구에 따라 미리 결정된다. 실시간 GPU 작업에 데드라인이 할당되면, 작업이 시작되고 데드라인까지 남은 시간을 계산하기 위해 작업 시작과 완료 시간을 추적해야 한다.

컨테이너에 데드라인이 할당되면 실제 컨테이너에서 실행되는 실시간 GPU 작업이 데드라인 이내에 실행이 되는지 확인해야 한다. 일반적으로 실시간 작업의 데드라인은 작업의 실제 실행 시간보다 더 많은 시간이 할당된다. 그리고 데드라인과 실제 작업 처리 시간은 다르기 때문에 실시간 작업이 데드라인 이내에 처리되는지 작업이 실행될 때마다 확인해야 한다. GPU 작업이 데드라인 이내에 실행되는지 확인하기 위해서는 GPU 작업의 실제 실행 시간을 확인하고 데드라인과 비교하여 해당 GPU 작업이 데드라인 이내에 처리를 완료하는지 확인해야 한다.

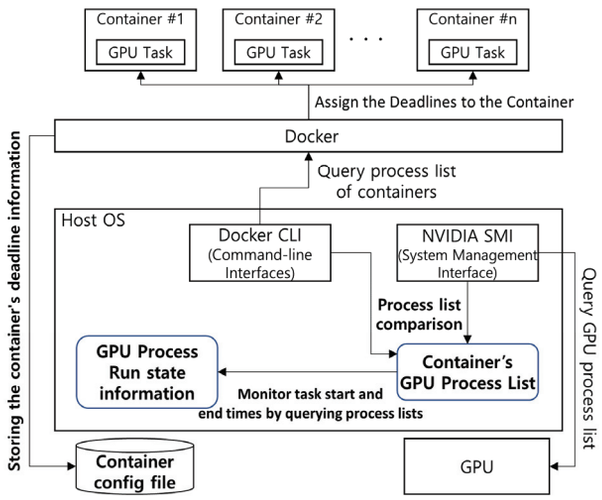

Fig. 6. Proposed Real-time GPU Task Run Status Monitoring System

```
CONTAINER ID  COMMAND    NAMES   DEADLINES
cfe038a11b18  "bash"     test5   2
e1762b8681bc  "bash"     test4   3
bb7b68c1a2b7  "bash"     test3   5
162d2dc519e0  "bash"     test2   1
584d18b4d14d  "bash"     test1   3
```

Fig. 7. List of Container with Deadline of Each GPU Tasks

```
var/lib/docker/containers/f42f1dcd329cdb240f13d
f42f1dcd329cdb240f13defe619a3ef236cfe8309cb3faf
cudacontainer00","Deadlines":"9","Driver":"over
0,"HasBeenStartedBefore":true,"HasBeenManuallyS
{},"SecretReferences":null,"ConfigReferences":n
containers/f42f1dcd329cdb240f13defe619a3ef236cf
```

Fig. 8. Deadline Contained in the Container Configuration File

본 논문에서는 실시간 GPU 작업의 작업 상태 모니터링을 위해 컨테이너 시스템인 Docker에 두 가지 기능을 추가한다. 첫 번째는 실시간 GPU 작업을 수행하는 컨테이너에 데드라인을 할당하고 컨테이너 설정 정보에 추가하기 위한 데드라인 관리 기능이고 두 번째는 실시간 GPU 작업의 시작 및 종료 시각과 남은 데드라인을 계산하기 위한 모니터링 기능이다.

상용 컨테이너 관리 시스템에는 데드라인을 관리하기 위한 기능을 제공하지 않는다. 컨테이너에 데드라인을 할당할 수 없으며, 실시간 GPU 작업의 데드라인을 계산하기 위한 기능을 제공하지 않는다. 본 논문은 실시간 GPU 작업을 지원하기 위해 오픈소스 기반 컨테이너 시스템인 Docker에 데드라인을 관리하기 위한 기능을 추가한다. 본 논문에서 제안하는 실시간 GPU 작업 실행 상태 모니터링 기법은 Fig. 6과 같다.

본 논문의 구현에서는 컨테이너에 데드라인을 할당할 수 있도록 Docker 엔진과 Docker CLI에 관련 기능을 추가한다. Docker는 컨테이너 관리자가 Docker CLI를 통해 컨테이너 생성, 삭제 및 모니터링과 같은 컨테이너 관리 명령을 요청하면 Docker 엔진에서 요청된 작업을 처리하는 서버-클라이언트 구조를 갖는다. 이러한 특성으로 인해 컨테이너에 데드라인을 할당하고 관리하기 위해서는 Docker 엔진과 Docker CLI에서 관련 정보를 교환하고 저장할 수 있도록 기능을 추가해야 한다. 본 논문에서는 Docker 엔진에서 컨테이너 설정 파일을 작성할 때 데드라인 정보를 저장할 수 있도록 컨테이너의 파라미터 정보를 추가하고 Docker CLI를 수정하여 컨테이너 관리자가 컨테이너 생성 및 모니터링 작업을 수행할 때 데드라인을 할당하고 확인하기 위한 기능을 추가한다.

본 논문에서는 컨테이너 생성 시 컨테이너에 데드라인을 할당하고 확인할 수 있도록 컨테이너 생성 명령어인 *docker create*와 *docker run*에 데드라인 정보를 입력하기 위한 옵션인 *-dl*을 추가하고 Fig. 7과 같이 컨테이너마다 할당된 데드라인을 확인할 수 있도록 실행 중인 컨테이너 리스트를 조회하기 위한 명령어인 *docker ps*에 컨테이너의 데드라인 정보를 출력할 수 있도록 기능을 추가하였다. 또한, Fig. 8과 같이 데드라인 정보가 영구 저장되도록 컨테이너가 생성될 때 Docker 엔진에서 작성하는 컨테이너 설정 파일인 config.v2.json 파일 양식에 데드라인 정보를 추가하도록 Docker 엔진을 수정하였다.

컨테이너에 데드라인이 할당되면 실제 컨테이너에서 실행되는 실시간 GPU 작업이 데드라인 이내에 실행되는지 모니터링해야 한다. 실시간 작업의 데드라인 보장 여부는 서비스 품질과 직접적으로 연관되어있는 사항이기 때문에 실시간 작업이 실행되고 데드라인 이전에 실행이 완료되는지 상시 추적할 수 있어야 한다. 앞서 설명한 것과 같이 실시간 작업의 데드라인은 일반적으로 작업의 실행 시간보다 더 많은 시간이 할당된다. 이로 인해 데드라인과 실제 작업의 실행 시간은 서로 다르다. 데드라인을 할당한다는 것은 작업이 완료되어야 하는 시간에 대한 기준을 정하는 과정일 뿐이며, 실제 작업이 데드라인 이내에 완료될 수 있다는 보장은 아니다.

실시간 작업의 데드라인 보장 상태를 확인하기 위해서는 데드라인, 작업 시작 및 종료 시각에 대한 정보가 필요하다. 사용자의 요청에 따라 작업이 시작되면 작업이 시작됨과 동시에 데드라인 타이머가 시작된다. 작업이 완료되면 데드라인 카운터가 종료되고 작업의 종료 시각과 데드라인을 비교하여 데드라인이 적중되었는지 실패했는지를 판단한다. 컨테이너 시스템은 컨테이너 관리자에 의해 운용되지만, 컨테이너에서 실행되는 작업은 서비스 제공자에 의해 운용된다. 이로 인해 컨테이너 관리자는 컨테이너에서 실행되는 작업의 시작 및 완료 시점을 알 수 없다. 컨테이너에서 실행되는 서비스를 개발할 때 작업 시작과 완료 상태를 전달하기 위한 기능을 추가하는 방법도 있지만 이러한 방법은 애플리케

이션의 투명성을 훼손하고 모니터링을 제3자가 제공하는 정보에 의존해야 하며, 이는 관리 측면에서의 신뢰성을 보장할 수 없다.

본 논문에서는 실시간 GPU 작업의 실행 상태를 추적하기 위해 프로세스 리스트 조회를 통해 컨테이너에서 실행하는 GPU 작업의 실행 상태를 추적한다. 컨테이너에서 실행되는 GPU 작업을 추적하기 위해서는 우선 GPU 작업의 실행 여부를 확인해야 한다. 또한, 컨테이너마다 할당된 데드라인이 서로 다르기 때문에 컨테이너별로 실행 중인 GPU 작업을 구분해야 한다. 본 논문에서는 각 컨테이너에서 실행되는 GPU 작업을 찾기 위해 GPU 프로세스 ID 리스트 정보를 사용한다.

앞서 설명한 것과 같이 GPU 모니터링과 컨테이너 모니터링은 별도의 시스템에서 관리된다. 이로 인해 컨테이너 모니터링 정보만 가지고는 컨테이너에서 실행되고 있는 프로세스가 어떤 작업인지 아닌지 알 수 없으며, GPU 모니터링 정보는 GPU 프로세스와 GPU 자원 사용량만 추적하기 때문에 해당 GPU 프로세스가 어떤 컨테이너에서 실행되는지 알 수 없다. 따라서 본 논문에서는 컨테이너에서 실행되는 프로세스 정보와 GPU에서 실행되는 GPU 작업의 프로세스 ID 정보의 비교를 통해 해당 GPU 작업이 어떤 컨테이너에서 실행되는지 식별한다.

새로운 GPU 작업이 실행되면 GPU 프로세스가 생성되고 GPU 프로세스 리스트에 추가되기 때문에 GPU 작업의 실행 상태는 비교적 간단하게 추적할 수 있다. 하지만, 각 컨테이너는 독립적인 개별 사용자가 관리하며, 컨테이너에서 실행되는 GPU 작업은 언제 실행될지 미리 알 수 없기 때문에 GPU 프로세스 리스트를 상시 조회하고 리스트의 변동을 확인해야 한다. 이로 인해 GPU 작업의 실행 상태를 추적하는 작업은 GPU 작업이 실제로 실행되고 있지 않아도 항상 프로세스 리스트를 조회하는 작업을 수행해야 한다. 본 논문에서 제안하는 GPU 작업 실행 상태 모니터링 기법은 Algorithm 1과 같이 작동한다.

Algorithm 1에서 보여주는 것과 같이 본 논문에서 제안하는 GPU 작업 실행 상태 모니터링 기법은 GPU 프로세스 리스트를 상시 조회하면서 새로 생성된 GPU 프로세스가 발견되면 GPU 작업이 발견된 시점을 작업 시작 시각으로 기록하고 컨테이너의 프로세스 리스트를 검사하여 어떤 컨테이너가 새로 생성된 GPU 작업을 수행하는지 찾는다. 또한, GPU 프로세스가 완료되어 프로세스 리스트에서 제외되면 작업 종료 시각으로 기록한다. 이러한 접근 방식을 통해 실시간 GPU 작업의 가장 최근 시작, 종료, 데드라인 여유시간을 측정하고 GPU 자원 사용량을 모니터링 한다.

제안하는 기법에서 실시간 GPU 작업을 실행하는 컨테이너를 찾고 작업의 시작 및 종료 시각 기록은 완전하게 사용자의 컨테이너 및 GPU 작업과 독립적으로 작동한다. 본 논문의 접근 방법은 컨테이너 사용자에게 어떠한 정보도 요구할

Algorithm 1: GPU task run state monitoring

```
1  struct GPUtaskinfo {
2    str ContainerID
3    str processID
4    int Deadline
5    float startTime
6    float endTime
7    bool deadlineHit
8    float floatTime
9  }
10 while()
11   get GPU process list;
12   if num of GPU process 〉 0
13     GPUtaskinfo->startTime = now()
14     GPUtaskinfo->Deadline = Deadline + now()
15     Get Container list
16     for 0 ~ num of Container
17       Get Container's process list
18         if Container's processID == GPU processID
19           struct GPUtaskinfo[] =
                    {ContainerID, processID,
                                Deadline, startTime}
20   if GPUtaskinfo != Previous GPUtaskinfo
21     SharedMemory = GPUtaskinfo[]
22     if remove Container n of list
23       free(GPUtaskinfo[n])
24     else
25       if remove GPU process n of list
26         GPUtaskinfo->endTime = now()
27         GPUtaskinfo->floatTime =
                      GPUtaskinfo->Deadline -
                          GPUtaskinfo->endTime
28         if GPUtaskinfo->floatTime 〈 0
29           GPUtaskinfo->deadlineHit = false
30         else
31           GPUtaskinfo->deadlineHit  = true
```

필요 없으며, GPU 작업을 개발할 때 실행 및 종료 시각을 전달받기 위한 추가 프로그래밍 작업도 필요 없다. 또한, 컨테이너와 GPU 모니터링 시스템의 프로세스 ID를 비교하는 간단한 방법을 사용하여 전체 시스템에 미치는 영향은 매우 작다. 본 논문이 제안하는 기법이 전체 시스템의 성능에 미치는 영향은 다음 장에서 자세히 설명한다.

5.2 실시간 GPU 작업 배치를 위한 노드 리스트 관리

클러스터로 구성된 클라우드 시스템을 관리할 때 가장 중요한 요소 중 하나는 부하분산이다. 이는 특정 노드에 사용자

가 집중되거나 특정 자원을 고용량으로 요구하는 사용자들이 하나의 노드에 집중되는 것을 방지하며, 이를 통해 특정 노드의 성능 저하와 클라우드 센터 사용자 사이의 성능 불균형을 방지하며, 이를 위해 일반적인 클라우드 환경에서 부하분산 작업은 각 노드의 자원 사용량과 사용자의 자원 요구량에 대한 정보에 초점을 두고 수행된다.

앞서 설명한 것과 같이 클라우드 환경에서 실시간 작업을 운용하는 경우 자원의 활용률보다 데드라인 위반 여부가 더 중요한 컨테이너 관리 메트릭이 된다. 실시간 작업을 운용하는 환경은 일반적인 클라우드 환경과같이 사용자 사이의 성능 영향이 발생할 요소를 최소화하여 성능 저하를 방지하고 빠르게 처리하는 것이 아니라 정해진 시간 이내에만 처리되면 되기 때문에 노드에서 실행 중인 모든 실시간 작업이 데드라인을 만족한다면 자원을 초과 사용하더라도 문제가 되지 않는다.

본 논문에서 설명한 것과 같이 본 논문의 목적은 노드에 최대한 많은 실시간 GPU 작업을 운용하면서 데드라인을 보장할 수 있도록 실시간 GPU 작업의 실행 상태를 추적하고 새로 실행된 컨테이너가 배치될 적절한 노드를 선택할 수 있도록 노드 리스트를 관리하는 것이다. 따라서, 본 논문에서는 일반적인 클라우드 환경과 다르게 GPU 작업의 실행 실패를 발생시키는 GPU 메모리 초과 사용을 방지하면서 노드에서 실행되고 있는 실시간 GPU 작업의 데드라인 정보와 데드라인 여유시간을 기반으로 컨테이너를 배치하기 위한 노드 리스트 관리기법을 제안한다.

앞서 설명한 것과 같이 실시간 GPU 작업의 데드라인은 실제 실행 시간보다 더 많은 시간이 할당된다. 실시간 GPU 작업이 시작되면 데드라인 시간까지 타이머가 작동하고 실시간 GPU 작업이 데드라인 이전에 완료되면 실제 작업 완료 시간과 데드라인 사이의 여유시간이 발생한다. 본 논문에서는 이러한 여유시간 정보를 기반으로 노드의 데드라인 여유시간을 계산하고 새로운 컨테이너 배치를 위한 메트릭으로 사용한다.

노드의 데드라인 여유시간이 높다는 것은 실시간 GPU 작업이 할당된 데드라인보다 더 빨리 완료된다는 것을 뜻하며, 이는 실시간 GPU 작업을 수행하는 컨테이너 사이에 데드라인 경쟁이 발생하지 않으면서 데드라인 위반 상황이 발생하지 않고 잘 운용되고 있다는 것을 나타낸다. 또한, 데드라인 여유시간이 높을수록 실시간 GPU 작업을 실행하는 새로운 컨테이너가 노드에 추가되어도 해당 노드에서 실행 중인 컨테이너들이 데드라인을 위반할 확률이 낮아진다. 일반적인 CPU 기반의 실시간 작업이라면 데드라인 여유시간을 기반으로 컴퓨팅 자원의 허용 범위 내에서 실시간 작업을 추가 실행할 수 있지만 본 논문과 같이 GPU 작업을 실시간 서비스 형태로 운용하기 위해서는 GPU 자원과 GPU 작업의 몇 가지 특성을 고려해야 한다.

GPU 작업은 일반적으로 GPU 메모리의 초과 사용을 허용하지 않는다. GPU 애플리케이션의 구현을 위해 가장 많이 사용되는 CUDA의 경우 일반적으로 GPU 연산을 위한 데이터를 GPU 메모리로 복사하기 위해 GPU 메모리 할당 작업을 수행하는데 *cudaMalloc* 함수를 사용한다. 해당 함수는 GPU 메모리의 초과 할당을 허용하지 않으며, 가용 GPU 메모리 용량보다 더 큰 데이터를 GPU 메모리로 복사하게 되면 GPU 메모리 부족으로 인한 작업 실패가 발생한다. CUDA는 GPU 메모리와 메인 메모리 사이의 통합 메모리 관리를 위해 GPU 메모리와 메인 메모리 사이에서 자동으로 데이터 스왑을 지원하는 *cudaMalloc Managed* 함수를 제공하지만 클라우드 관리자는 컨테이너에서 실행되는 GPU 작업의 GPU 메모리 할당 작업이 어떤 형태로 구현되어있는지 알 수 없기 때문에 실시간 GPU 작업의 실패를 방지하기 위해서는 가용 GPU 메모리의 용량을 고려해야 한다.

또한, 이전 장에서 수행한 실험에서 보여주는 것과 같이 GPU 코어의 사용량이 모니터링 정보상 100%가 된 상태에서 추가 GPU 작업을 실행하면 실행 중인 모든 GPU 작업의 성능이 저하되는 것을 확인할 수 있었다. 특히, GPU의 경우 CPU나 메모리와 같은 컴퓨팅 자원과는 다르게 자원의 확장성에 제한이 있기 때문에 자원 고갈이 더 빠르게 발생한다. GPU의 모니터링을 위한 NVIDIA-SMI의 경우 GPU 코어 사용량을 100%까지만 측정할 수 있기 때문에 GPU 코어 사용량이 100%에 도달하면 실제로 100%를 사용하는 것인지 얼마나 초과 사용되고 있는 것인지 알 수 없다.

이러한 특성으로 인해 본 논문에서는 GPU 코어 사용량이 100% 미만인 경우에만 컨테이너를 배치하기 위한 노드의 후보로 선택한다. 이전 장에서 실험한 것과 같이 GPU 코어 사용량이 100%인 경우 새로운 컨테이너가 몇 개 더 실행되는 것으로 인한 큰 성능 영향이 없지만, 실제 실시간 GPU 작업을 운용할 때 사용자의 GPU 작업 규모를 알 수 없기 때문에 GPU 코어 사용량이 100% 미만인 노드만 새로운 컨테이너를 배치하기 위한 노드의 후보로 결정한다.

일반적으로 GPU 메모리는 초과 사용을 허용하지 않기 때문에 GPU 메모리가 초과 사용되는 경우 앞서 설명한 GPU 코어와 같이 성능 저하 수준의 문제가 아니라 작업 실행 실패로 이어진다. 메인 메모리의 경우 가상 메모리 기술을 활용해 실제 물리적인 메모리 용량보다 더 큰 데이터를 할당할 수 있다. 메인 메모리를 초과 사용하는 경우 성능 저하를 동반하면서 가상 메모리와 실제 물리 메모리 사이의 데이터 스왑을 통해 초과 사용을 가능하게 한다. 하지만 GPU 메모리의 경우 OOM(Out of Memory) 에러가 발생하면 작업 실행에 실패하게 된다. 컨테이너의 GPU 메모리 사용량은 실제 GPU 작업이 실행되기 전에 알 수 없다. 따라서 본 논문에서는 가용 GPU 메모리 용량의 크기에 따라 내림차순으로 정렬하여 새로운 컨테이너를 배치할 후보 노드 리스트를 관리한다. 본 논문에서 제안하는 GPU 자원 사용량과 데드라인 여유시간을 기반으로 작동하는 컨테이너 배치를 위한 노트 리스트 관리기법은 Algorithm 2와 같이 작동한다.

Algorithm 2: Node List Ranking

```
   Worker Node Side
1  struct nodeInformation {
2    int nodeState = 0 or 1
     //1: Provisionable, 0: Unprovisionable
3    float nodeFloattime
4    int GPUmemusage
5  }
6  if request information from Master node
7    for 0 ~ num of Container
8      if deadlineHit == false
9        hitFalse++
10     if (hitFalse 〉 0 || GPU core usage == 100%
                        || GPU memory usage == 100%)
11       nodeState = 0
12     else
13       for 0 ~ num of Container
14         nodeFloattime=+ GPUtaskinfo->floatTime
15         nodeState = 1
16     send nodeInformation

   Master Node Side
1  if request new container create
2    request Node information
3    sort list of nodes based on available GPU memory
4    for each node list
5      if new container's deadline 〈 nodeFloattime
6        candidateNode = node n
7        break
```

Algorithm 2와 같이 본 논문에서 제안한 기법은 기본적으로 가용 GPU 메모리 용량을 기반으로 컨테이너 배치를 위한 후보 노드의 순위를 결정한다. 가용 GPU 메모리 용량은 실시간 GPU 작업의 데드라인 보장 이전에 작업 실패와 연관된 메트릭이기 때문에 가장 처음 가용 GPU 메모리 용량을 기반으로 컨테이너 배치를 위한 후보 노드의 순위를 정한다. 가용 GPU 메모리의 용량이 충분하다는 것은 OOM 에러가 발생할 확률이 낮다는 것을 뜻하며, 실제 실행 중인 실시간 GPU 작업이 데드라인을 보장할 수 있다는 것은 아니다. 따라서, 가용 GPU 메모리 용량이 높은 순서대로 후보 노드의 리스트를 관리하면서 각 노드의 데드라인 여유시간을 확인한다.

앞서 설명한 GPU 작업 실행 상태 모니터는 GPU 프로세스 정보를 수집하면서 GPU 자원 사용량에 대한 정보를 수집한다. 수집된 모니터링 정보를 기반으로 Algoithm 2와 같이 클러스터를 구성한 노드 리스트는 전역 리스트로 관리된다. 실시간 GPU 작업을 실행하는 새로운 컨테이너가 생성되면 컨테이너 생성 시 입력된 데드라인 정보를 기반으로 새로운 컨테이너를 배치할 때 가용 GPU 자원이나 데드라인 여유시간이 가장 많은 노드가 리스트에서 가장 높은 순위를 가지며, 새로운 컨테이너 배치가 요청되면 가장 순위가 높은 노드를 선택한다.

본 논문에서 제안하는 노드 리스트 관리기법의 목적은 클러스터 기반 컨테이너 환경에서 새로운 컨테이너를 배치할 노드를 선택할 때 참조하기 위한 후보 노드의 순위를 결정하는 것이다. 또한, 노드 리스트를 구축하기 위한 정보를 최소화하여 모니터링 정보 수집 및 통합에 대한 오버헤드를 감소시키는 것이다. 본 논문에서는 GPU 작업의 특성을 기반으로 노드 리스트 구축을 위한 모니터링 메트릭의 우선순위를 설정하였고 이에 따라 필수적인 부분만 수집한다. 또한, 마스터 노드로 전송하는 정보를 최소화하기 위해 자원 사용량과 데드라인 여유시간을 기반으로 배치 가능 여부를 결정하고 마스터 노드에 배치 가능 여부와 데드라인 여유시간 정보만을 전송한다.

GPU 작업은 일반적인 CPU 기반 연산 작업보다 CPU를 사용한 연산 부분이 상대적으로 작다. 대부분의 연산은 GPU에서 처리되기 때문에 CPU에 대한 정보의 중요성이 낮다. 또한, 메인 메모리나 스토리지의 경우 컴퓨팅 자원 중에 비용이 가장 낮고 확장성이 매우 높으며, CPU나 GPU보다 비교적 가용 용량이 여유롭고 기아 상태에 빠질 확률이 낮으며, 상용 GPU 서버에서 일반적으로 GPU 메모리 용량보다 더 많은 용량이 설치되기 때문에 노드 리스트 구축을 위한 메트릭에서 제외한다.

본 논문에서 기본적으로 새로운 컨테이너를 배치하기 위한 노드 리스트는 단일 전역 리스트로 관리되며, 노드를 선택할 때 새로 실행되는 컨테이너의 데드라인 값에 따라 최적화된 리스트 정렬 작업은 수행하지 않는다. 컨테이너에 할당된 데드라인 정보에 따라 가장 적절한 여유시간을 가진 노드에 배치하는 것은 추후 연구에서 진행할 것이다.

6. 실 험

이번 장에서는 실험을 통해 본 논문에서 제안하는 실시간 GPU 실행 상태 모니터와 노드 리스트 관리기법의 성능을 측정하고 전체 시스템에 미치는 영향을 분석한다. 본 논문에서 제안한 기법은 앞서 설명한 것과 같이 크게 세 가지 서브 시스템으로 구성된다. 그 중, 전체 시스템에 성능 영향을 주는 실시간 GPU 작업 상태 모니터링 기법과 새로운 컨테이너 배치를 위한 노드 리스트 관리기법에서 각 노드의 모니터링 정

보를 받고 노드 리스트를 구성하는 작업의 성능을 측정한다. 이 두 가지 서브 시스템은 컨테이너와 GPU의 모니터링 정보를 추출하고 비교하는 작업과 시간 측정을 위한 작업을 수행하며, 새로운 컨테이너 배치를 위한 노드 리스트 구성 작업은 모든 노드의 모니터링 정보를 수집하여 처리하기 때문에 기존 환경과 비교하여 컴퓨팅 자원을 추가로 사용하며, 작업 실행으로 인한 오버헤드를 발생시킨다. 따라서, 이번 장에서는 성능 측정을 통해 본 논문에서 제안하는 기법의 자원 사용량과 성능을 측정한다. 실험 환경은 Table 1과 같다.

실험에서는 10개의 컨테이너를 사용하며, 각 컨테이너는 GPGPU 프로그래밍 모델인 CUDA를 사용해 구현한 5,120×5,120 행렬 곱셈 작업을 반복적으로 수행한다. 각 컨테이너에서 실행하는 GPU 작업의 실행 시간을 계산하기 위해 컨테이너에서 실행 중인 프로세스의 ID와 GPU 모니터링 도구를 사용해 추출한 GPU 프로세스 ID를 비교하는 작업과 GPU 작업의 실행 시간을 계산하는 작업의 성능을 측정한다. 실험 결과는 Fig. 9와 같으며, 본 논문에서 제안한 실시간 GPU 작업 실행 상태 모니터링 작업이 사용하는 자원 사용량은 Fig. 10에서 보여준다.

실험 결과에서 보여주는 것과 같이 컨테이너에서 실행 중인 GPU 작업의 실행 상태를 추적하기 위한 모니터링 작업은 실행 간격에 대한 시간적 제한 없이 지속적으로 모니터링 정보를 조회하고 수집한다. 이를 통해 임의의 시간에 실행되는 GPU 작업을 찾고 실행 상태를 추적할 수 있다. Fig. 8과 같이 본 논문에서 제안하는 GPU 작업 실행 상태 모니터는 약 0.2초 간격으로 GPU 프로세스 리스트를 조회하며, 프로세스 리스트를 확인하고 실행 중인 GPU 작업이 프로세스 리스트에서 제거되면 GPU 작업이 종료된 것으로 판단하여 해당 GPU 작업의 종료 시각을 기록한다. Table 2는 GPU 프로그램의 소스코드에 시간 측정을 위한 타이머 함수를 사용했을 때와 본 논문에서 제안한 GPU 작업 실행 상태 모니터링 기법에서 측정한 GPU 작업의 시작 및 종료 시각을 비교한 것이다. 실험에서는 임의의 시간에 GPU 작업을 실행했을 때 모니터링 측정값과 실제 실행 시각의 오차를 측정한다.

앞서 수행한 실험 결과에서 보여주는 것과 같이 본 논문에서 제안하는 GPU 작업 실행 상태 모니터의 경우 컨테이너 모니터링 정보와 GPU 모니터링 정보의 통합 작업을 별도의 시간 간격 없이 지속적으로 수행한다. 이로 인해 Table 2의 실험 결과와 같이 GPU 작업의 실제 실행 시간과 모니터링 측정값의 오차는 모니터링 작업이 실행되는 시간 이하로 매우 작게 발생한다. 또한, 논문에서 제안하는 방법은 컴퓨팅 자원 사용량이 매우 낮아서 전체 시스템에 미치는 영향은 거의 없다. 특히, 본 논문에서 대상으로 하는 실시간 GPU 작업의 경우 작업의 대부분을 GPU로 처리하지만 본 논문에서 제안하는 실시간 GPU 작업 상태 추적 모니터링 기법은 CPU와 메모리를 사용하기 때문에 GPU 작업에 미치는 영향은 거의 없다. 또한, 사용자 컨테이너와 컨테이너에서 실행되는 GPU 작업과 완전하게 독립적으로 작동하기 때문에 실시간 GPU 작업의 규모와 상관없이 일관적인 모니터링 작업 성능을 달성한다.

다음 실험은 새로운 컨테이너를 배치할 때 마스터 노드가 워커 노드에게 요청하는 모니터링 정보의 용량을 측정하고 노드의 개수에 따른 노드 리스트 구성 시간을 측정한다. Table 3은 새로운 컨테이너 배치 시 배치 후보 노드의 리스

Table 1. Experiment Environment

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

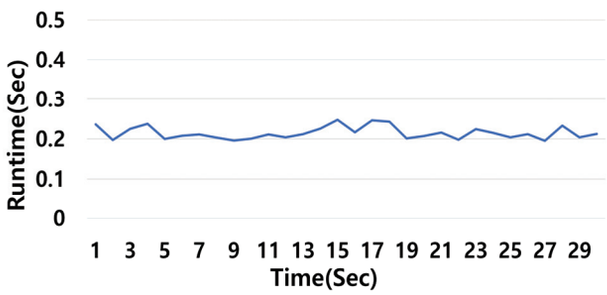

Fig. 9. Performance of Real-time GPU Task Run Status Monitoring System

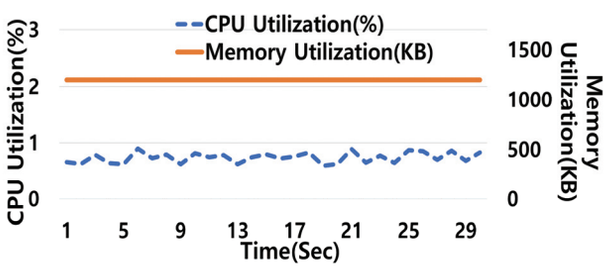

Fig. 10. Resource Usage of the Real-time GPU Task Run Status Monitoring System

Table 2. Actual Execution Time of GPU Tasks and Measured Time Through GPU Tasks Execution Status Monitoring

	Difference between actual measured time and monitoring result(Sec)
Start time	+0.22 (min: +0.18, max: +0.25)
End time	+0.21 (min: +0.18, max: +0.24)
Runtume	+0.22 (min: +0.17, max: +0.24)

Table 3. Monitoring Information for Node List Configuration

Tpye	Node state	GPU memory usage	Deadline float time
Size	1 Byte	5 Byte	10 Byte

트를 구성하기 위해 각 노드로부터 전송받는 모니터링 정보의 용량을 보여준다.

Table 3과 같이 본 논문에서 제안한 노드 리스트 관리기법은 최소한의 정보만으로 실시간 GPU 작업을 실행하는 새로운 컨테이너를 배치할 수 있는 노드의 순위를 결정한다. 앞서 설명한 것과 같이 GPU 작업을 운용할 때 중요성이 높은 자원 정보 메트릭을 반영함으로써 불필요한 정보의 전송을 방지한다. 이로 인해 마스터 노드에서 컨테이너 배치를 위해 수집하는 정보는 감소하고 새로운 컨테이너 배치를 위한 모니터링 정보 전송 작업의 네트워크 대역폭 사용량도 줄어들게 된다. 새로운 컨테이너가 임의의 시간에 생성되고 종료되는 클라우드 환경을 관리하기 위해서는 모니터링이 필수적이다. 이 과정에서 모니터링 작업과 사용자의 작업이 노드의 컴퓨팅 자원을 같이 사용하게 되는데 모니터링 정보의 간소화는 모니터링으로 인한 자원 사용을 감소하기 때문에 사용자에게 끼치는 영향을 최소화할 수 있다.

다음 실험은 각 노드로부터 수집된 모니터링 정보를 새로운 컨테이너 배치를 위한 노드 리스트로 구성하는 작업의 성능을 측정한다. 이번 실험은 시뮬레이션을 통한 성능 측정을 수행하며, 노드 정보는 본 논문에서 제안한 기법과 동일한 형식으로 임의의 값이 저장된 가상 데이터를 생성하고 각 노드에 대한 모니터링 정보의 증가, 즉, 노드의 개수가 증가함에 따라 노드 리스트를 구성하는데 소요되는 시간을 측정한다. 실험 결과는 Fig. 11에서 보여준다.

Fig. 11의 실험 결과에서 보여주는 것과 같이 각 노드로부터 전송받은 데이터를 가용 GPU 메모리 용량에 따라 노드 리스트를 구성하는 작업은 실행 시간이 매우 짧다. 실험 결과에서 보여주는 것과 같이 노드가 2개에 대한 데이터와 200개에 대한 데이터로 노드 리스트를 구성할 때 각각 157ms와 164ms의 시간이 소요되며, 이는 노드 리스트를 구성하는 작업 성능의 변동량은 크지 않는 것을 보여준다. 또한, 컨테이너를 배치할 수 없는 노드의 경우 모니터링 정보를 전송하지 않기 때문에 해당 노드는 노드 리스트 구성 대상에서 제외되며, 이로 인해 불필요한 작업을 최소화한다. 또한, 서버리스 환경과 같이 작업의 요청과 동시에 컨테이너가 생성되는 환경에서 새로 생성되는 컨테이너가 배치될 노드를 찾는 작업은 컨테이너의 작업 시간에 포함된다. 본 논문에서 제안하는 기법은 최소한의 정보로 컨테이너 배치를 위한 후보 노드의 리스트를 구성하기 때문에 실시간 작업의 실행 시간에 미치는 영향은 매우 적다.

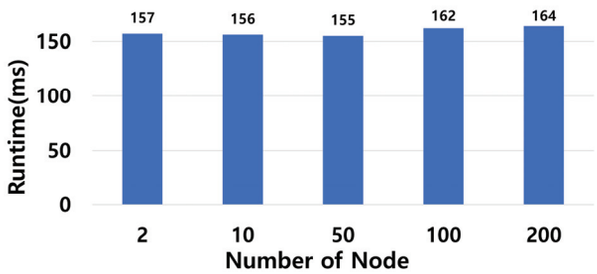

Fig. 11. Node List Configuration Performance with Simulations and Dummy Data

앞서 수행한 실험 결과에서 보여주는 것과 같이 본 논문에서 제안한 실시간 GPU 작업 실행 상태 모니터와 노드 리스트 관리기법은 전체 시스템에 큰 영향을 미치지 않는다. 모니터링 작업은 매우 작은 용량의 컴퓨팅 자원만 사용하며, GPU를 사용한 연산 작업을 수행하지 않기 때문에 실시간 GPU 작업에 미치는 영향은 없다. 또한, 새로운 컨테이너를 배치하기 위한 노드 리스트 관리기법의 경우 각각의 노드에서 자원 사용량이나 데드라인 여유시간을 기반으로 새로운 컨테이너의 배치 가능 여부를 결정하기 때문에 마스터 노드는 배치 가능 여부에 대한 정보와 GPU 메모리 용량, 데드라인 여유시간과 같이 3가지 정보만 받아오면 되기 때문에 일반적으로 클라우드 환경에서 노드 관리를 위해 수집되는 CPU와 메모리 사용량, 노드에서 실행 중인 컨테이너 혹은 가상머신의 개수와 같이 다양한 메트릭을 포함한 모니터링 정보를 필요로 하지 않는다. 모니터링 정보의 간소화는 모니터링 정보 전달로 인한 네트워크 오버헤드를 감소시킬 수 있으며, 새로운 컨테이너 배치를 위한 사전 작업을 단순화할 수 있다.

7. 결 론

본 논문에서는 실시간 GPU 작업을 운영하기 위해 오픈소스 기반 컨테이너 시스템인 Docker에서 컨테이너에 데드라인 정보를 할당할 수 있도록 데드라인 관리 기능을 추가하고 사용자의 컨테이너와 GPU 작업으로부터 어떠한 정보도 필요로 하지 않으면서 독립적으로 작동하는 실시간 GPU 작업 실행 상태 모니터와 컨테이너 배치를 위한 노드 리스트 관리기법을 제안했다. 실험 결과에서 보여주는 것과 같이 본 논문에서 제안한 기법은 전체 시스템에 큰 영향을 미치지 않으며, 본 논문에서 제안한 기법으로 인한 컴퓨팅 자원 사용량은 매우 작기 때문에 노드에서 실행 중인 컨테이너와 컴퓨팅 자원 경쟁을 하지 않는다. 특히, 본 논문에서는 실시간 GPU 작업을 대상으로 하며, 제안된 기법은 GPU를 사용한 추가 작업을 수행하지 않기 때문에 연산 작업의 대부분을 GPU에서 처리하는 실시간 GPU 작업에 영향을 주지 않는다.

이를 통해 실시간 GPU 작업을 실행하는 컨테이너에 데드라인 할당 및 관리를 할 수 있으며, GPU 작업의 실행 상태를 실시간으로 추적하여 데드라인 위반 여부를 런타임 시간에 알 수 있다. 또한, 이전 장에서 설명한 것과 같이 본 논문에서는 데드라인 상태를 노드 관리를 위한 가장 중요한 메트릭으로 초점을 두고 새로운 컨테이너를 배치하기 위한 노드 리스트는 데드라인 여유시간을 기반으로 후보 노드의 리스트를 관리하며, 모니터링 정보 전송은 데드라인 및 GPU 자원 상태에 따라 각 노드가 결정하기 때문에 불필요한 모니터링 정보 전송을 최소화할 수 있다.

앞서 설명한 것과 같이 본 논문에서 제안한 기법은 Docker 환경에서 데드라인 정보 관리, 실시간 GPU 작업 실행 상태 모니터링 그리고 새로운 컨테이너 배치를 위한 노드 리스트 관리에 초점을 둔다. 또한, 자원 활용률의 향상보다는 데드라인 위반 방지에 초점을 두기 때문에 새로운 컨테이너를 배치하기 위한 노드 리스트를 관리할 때 데드라인 여유시간의 크기에 따라 노드 리스트의 순위를 관리한다. 본 논문의 추후 연구에서는 제안된 기법을 기반으로 클러스터 환경에서 새로운 컨테이너 배치 및 실행 작업이 가능하도록 추가 개발을 수행하고 데드라인 위반 방지뿐만 아니라 자원 활용률 향상을 위해 데드라인 여유시간과 자원 사용량을 기반으로 새로 생성되는 컨테이너의 데드라인에 적합한 best-fit 배치 기법을 연구할 것이다.

References

[1] J. H. Kang and J. M. Gil, “Deadline information management techniques to support real-time GPU tasks in container-based cloud environments,” *Proceedings of the Annual Spring Conference of Korea Information Processing Society Conference (KIPS) 2022*, Vol.29, No.1, pp.56-59, 2022.

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

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

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

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

[6] Docker, docker stats [Internet], https://docs.docker.com/engine/reference/commandline/stats/

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

[8] The Linux Foundation, Prometheus [Internet], https://prometheus.io/

[9] Google, cAdvisor [Internet], https://hub.docker.com/r/google/cadvisor/

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

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

[12] J. Ru, Y. Yang, J. Grundy, J. Keung, and L. Hao, “An efficient deadline constrained and data locality aware dynamic scheduling framework for multitenancy clouds,” *Concurrency and Computation: Practice and Experience*, Vol.33, No.5, e6037, 2021.

[13] J. Lou, Z. Tang, S. Zhang, W. Jia, W. Zhao, and J. Li, “Cost-effective scheduling for dependent tasks with tight deadline constraints in mobile edge computing,” *IEEE Transactions on Mobile Computing* (Early Access), 2022.

[14] H. Xia, M. Liu, Y. Chen, X. Jin, Z. Wang, and F. Wang, “A load balancing strategy of container virtual machine cloud microservice based on deadline limit,” *14th International Conference on Measuring Technology and Mechatronics Automation (ICMTMA)*, pp.998-1002, 2022.

[15] V. Struhár, S. S. Craciunas, M. Ashjaei, M. Behnam, and A. V. Papadopoulos, “React: Enabling real-time container orchestration,” *26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp.1-8, 2021.

[16] C. Singh, P. Kumari, R. Mishra, H. P. Gupta, and T. Dutta, “Secure industrial iot task containerization with deadline constraint: A stackelberg game approach,” *IEEE Transactions on Industrial Informatics* (Early Access), 2022.

[17] L. Ye, Y. Xia, L. Yang, and C. Yan, “SHWS: Stochastic Hybrid Workflows Dynamic Scheduling in Cloud Container Services”, *IEEE Transactions on Automation Science and Engineering*, Vol.19, No.3, pp.2620-2636, 2021.

[18] K. Dubey and S. C. Sharma, “A novel multi-objective CR-PSO task scheduling algorithm with deadline constraint in cloud computing,” *Sustainable Computing: Informatics and Systems*, Vol.32, 100605, 2021.

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

[20] Yann LeCun, Corinna Cortes, and Chris Burges, MNIST handwritten digit database [Internet], http://yann.lecun.com/exdb/mnist/

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

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

길 준 민

https://orcid.org/0000-0001-6774-8476
e-mail : jmgil@cu.ac.kr
1994년 고려대학교 전산학과(학사)
1996년 고려대학교 전산학과(석사)
2000년 고려대학교 전산과학과(박사)
2001년 ~ 2002년 University of Illinois at Chicago, Post-Doc.
2002년 ~ 2006년 KISTI 슈퍼컴퓨팅센터 선임연구원
2006년 ~ 2010년 대구가톨릭대학교 컴퓨터교육과 교수
2010년 ~ 현 재 대구가톨릭대학교 컴퓨터소프트웨어학부 교수
관심분야 : 클라우드컴퓨팅, 빅데이터, 인공지능, 분산시스템