논문 20-23-01-06 KNOM Review '20-01 Vol.23 No.01
https://doi.org/10.22670/knom.2020.23.1.43

GPU 컨테이너 동시 실행에 따른 응용의 간섭 측정 프레임워크 설계

김 세 진◆, 김 윤 희°

A design of GPU container co-execution framework measuring interference among applications

Sejin Kim◆, Yoonhee Kim°

요 약

범용 그래픽 처리 장치(General Purpose Graphics Processing Unit, GPGPU)는 최근 고성능 컴퓨팅에서 중요한 역할을 함으로써, 여러 클라우드 서비스 공급업체들은 GPU 서비스를 제공하기 시작했다. 컨테이너를 사용하는 클라우드 환경에서 대부분의 클러스터 오케스트레이션 플랫폼은 정수 개의 GPU를 작업에 할당하고 다른 작업과 이를 공유하는 것을 허용하지 않는다. 이 경우 작업이 GPU에서 코어 및 메모리 등 자원이 집중적으로 필요하지 않다면 GPU 노드의 리소스 사용률이 저하될 수 있다. GPU 가상화는 응용의 동시 수행을 가능하게 하며 자원을 공유할 수 있는 기회를 제공한다. 하지만 응용의 동시 수행 성능은 동시 수행되는 응용의 특성과 노드 안에서 자원 경쟁으로 인한 간섭에 따라 달라질 수 있다. 본 논문은 컨테이너 오케스트레이션 플랫폼인 쿠버네티스(Kubernetes)를 기반으로 다중 서버 생성 및 실행을 통하여 GPU를 공유함으로써 발생할 수 있는 간섭을 측정하기 위한 프레임워크를 제안한다. 본 프레임워크를 통해 다양한 스케줄링 방법으로 GPU에서 여러 작업을 실행함으로써 이에 따른 성능 변화를 조사하였으며, 이를 통해 GPU 메모리 사용량 및 컴퓨팅 리소스만 고려해서는 최적의 스케줄링을 할 수 없음을 보인다. 마지막으로 해당 프레임워크를 사용하여 응용들의 동시 실행에 따라 발생한 간섭을 측정한다.

Key Words : GPU, container, co-execution, cluster orchestration framework, interference

ABSTRACT

As General Purpose Graphics Processing Unit (GPGPU) recently plays an essential role in high-performance computing, several cloud service providers offer GPU service. Most cluster orchestration platforms in a cloud environment using containers allocate the integer number of GPU to jobs and do not allow a node shared with other jobs. In this case, resource utilization of a GPU node might be low if a job does not intensively require either many cores or large size of memory in GPU. GPU virtualization brings opportunities to realize kernel concurrency and share resources. However, performance may vary depending on characteristics of applications running concurrently and interference among them due to resource contention on a node. This paper proposes GPU container co-execution framework with multiple server creation and execution based on Kubernetes, container orchestration platform for measuring interference which may be occurred by sharing GPU resources. Performance changes according to scheduling policies were investigated by executing several jobs on GPU. The result shows that optimal scheduling is not possible only considering GPU memory and computing resource usage. Interference caused by co-execution among applications is measured using the framework.

※이 성과는 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No. 2020R1H1A2011685)
◆ First Author : Sookmyung Women's University, wonder960702@gmail.com
° Corresponding Author : Sookmyung Women's University, yulan@sookmyung.ac.kr
논문번호 : KNOM2020-01-11, Received July 17, 2020; Revised July 30, 2020; Accepted August 16, 2020

Ⅰ. 서 론

GPGPU(General-Purpose Graphic Processing Unit)는 지난 몇 년 동안 컴퓨팅 가속 처리 제공으로 ML(Machine Learning), HPC(High Performance Computing) 응용 등 다양한 분야에서 폭넓게 사용된다. 이러한 이유로 클라우드 및 서버 인프라에서는 사용자의 다양한 응용 프로그램의 실행을 위해 GPU 서버를 제공하고 있다. Amazon EC2 [1], Nimbix [2], Microsoft Azure [3], Alibaba [4] 와 같은 여러 대규모 클라우드 회사는 GPU 서비스를 제공한다. 클라우드 서비스 환경에서 높은 이식성과 유연성 및 효율성으로 주목받고 있는 컨테이너는 응용을 실제 구동 환경으로부터 추상화할 수 있는 패키징 메커니즘을 제공한다. 컨테이너로써 응용 전체를 패키지로 묶어 운영체제와 코드 자체까지 추상화하여 응용의 배포 및 전반적인 관리가 쉬워지므로 퍼블릭 클라우드뿐 아니라 프라이빗 클라우드에서도 인기를 확장하고 있다 [5].

클러스터 오케스트레이션 플랫폼인 쿠버네티스는 오케스트레이션 엔진을 통해 컨테이너의 생성과 소멸, 시작과 중단 시점 제어, 스케줄링, 로드 밸런싱 등 컨테이너로 응용을 구성하는 모든 과정을 관리할 수 있다. CPU 및 메모리 리소스를 처리하는 쿠버네티스의 작업 스케줄러는 다양한 작업을 사용자가 명시한 정책에 따라 적절한 리소스에 할당한다 [6]. 하지만 GPU에서 워크로드를 실행하는 경우, 쿠버네티스는 GPU를 확장된 리소스로써 인식하므로 스케줄러는 요청한 GPU개수만큼만 리소스를 할당해준다. 이는 워크로드가 소비하는 GPU 활용도 및 메모리 사용량에 상관없이 정수개의 GPU에만 작업을 할당할 수 있다.

그림 1의 왼쪽 부분이 해당 문제를 자세히 설명하고 있다. 1개의 GPU가 있는 2개의 노드와 다양한 유형을 가진 네 종류 (j_0, j_1, j_2, j_3) 의 작업이 두 개씩 존재하여 총 8개 작업이 있으며 각 작업의 수행시간이 동일하다고 가정한다. 작업이 각 노드에 할당되자마자 컨테이너가 실행된다. 그림에서 알 수 있듯이 쿠버네티스 디폴트 스케줄러[6]을 사용할 때 총 실행시간은 4t이며 각 노드의 작업이 적은 양의 리소스를 소비하더라도 하나의 노드에 하나의 작업만 할당할 수 있기 때문에 물리적 리소스가 낭비될 수 있다. 적은 양의 자원이 필요한 작업이 제출될수록 더 많은 자원이 낭비된다.

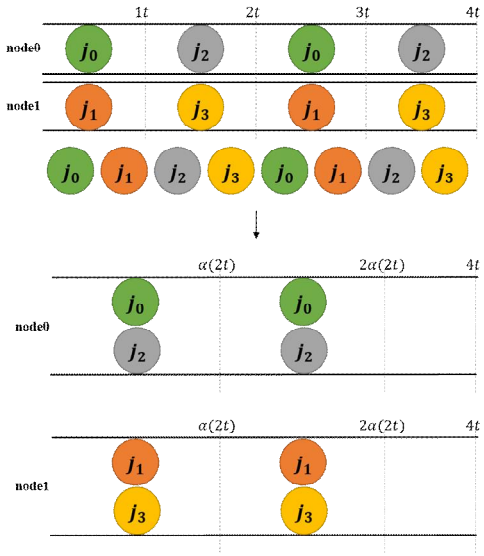


그림 1. 기본 및 자원 공유 스케줄링의 작업 배치
Fig. 1. Placement of the job with default and resource sharing scheduling

본 논문은 스케줄링 메커니즘의 문제점을 해결하기 위해 다중 서버 생성 및 실행을 통한 노드의 자원 공유 환경을 제안한다. 그림 1의 아래쪽 부분에서 볼 수 있듯이 각 작업은 노드 당 생성된 다중 서버에 배포 및 실행된다. 작업은 노드의 GPU 자원을 공유할 수 있으므로 각 작업의 전체 실행 시간 및 대기 시간이 줄어든다. 자원 경쟁으로 인한 오버헤드는 발생하지만 기존 방법의 4t보다 실행 시간이 감소한다. 따라서 GPU 공유를 통한 작업 할당은 총 실행 시간을 줄이고 GPU 자원 활용도를 향상시킬 수 있다. 또한, GPU 자원 공유 환경에서 실험을 통해 동시 실행할 작업 배치의 중요성을 보이고 스케줄링 전략에 따른 작업 배치 실험을 통해 GPU 메모리 사용량 및 컴퓨팅 리소스만 고려해서는 최적의 스케줄링을 할 수 없음을 보인다. 작업 배치 실험 결과를 토대로 GPU 자원 공유 환경에서 여러 자원의 경쟁으로 인해 간섭이 발생함을 보이고, 본 논문에서 제안한 프레임워크를 사용하여 이를 측정한다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구를 소개하며, 3장에서는 시스템 디자인을 설명한다. 해당 시스템을 사용한 실험 결과는 4장에서 보여주며 5장을 통해 결론을 짓는다.

Ⅱ. 관련 연구

여러 컨테이너에 GPU 리소스를 효과적으로 분배하기 위해서는 GPU 공유 방법이 필요하다. 하지만 클러스터 오케스트레이션 플랫폼인 쿠버네티스에는 클러스터에서 GPU 공유 방법을 적용하는 데에 몇 가지 문제가 존재한다.

- 쿠버네티스 마스터 노드는 GPU 노드의 리소스 사양을 인식하지 못한다.
- GPU는 비선점 자원으로써, 적은 리소스를 소비하는 긴 작업은 효율적인 리소스 사용률을 얻을 수 없다.

[7] 논문은 다양한 워크로드에 적용할 수 있는 유연한 컨테이너 기반 스케일링 정책을 제안했다. 컨테이너의 수평 및 수직 스케일링을 통해 워커노드를 관리하며, 컨테이너 확장성에 대한 정책으로써 RL (Reinforcement Learning)을 도입하여 스케일을 자동으로 조정하는 클라우드 정책을 도입했다. 하지만, GPU 컨테이너에서는 수직 확장이 불가능하며, 해당 논문에서 사용된 컨테이너 클러스터 플랫폼인 Docker swarm은 여러 개의 가상 서버로 구성할 수 없다.

[8,9]는 쿠버네티스 환경에서 GPU 공유를 위한 스케줄링 및 프레임워크를 개발했다. GPU와 GPU 메모리를 공유하기 위해 물리적 GPU를 여러 개의 가상 GPU로 나누었으며, 사용자가 요청한 양만큼 가상 GPU를 컨테이너에 할당한다. 리소스 사용률을 높이기 위해 동적 리소스 할당을 채택했으나, 제안된 정책은 리소스 사용패턴이 유사한 ML(Machine Learning) 응용으로 제한된다. 또한, 실제 GPU를 고려하지 않고 단순히 여러 개의 가상 GPU를 생성하므로 워크로드 실행 시 실제 실행 환경의 특성이 반영되지 않을 수 있다.

Ⅲ. 시스템 디자인

1. 동시 실행에 따른 간섭 측정 시스템 디자인

그림 2는 본 논문에서 제안하는 GPU 공유 컨테이너 클러스터의 전반적인 구조를 보여준다. 제안하는 시스템 구조는 하나의 마스터 노드와 n개의 워커 노드로 구성된다.

GPU 공유 컨테이너 클러스터는 호스트 시스템의 운영체제 위에서 작동한다. 온 디맨드 클라이언트의 CPU, DRAM, 디스크, 네트워크, GPU 등의 하드웨어 자원을 격리한다. 스케줄링 모듈, 리소스 관리 모듈 및 리소스 모니터링 모듈을 포함하는 마스터

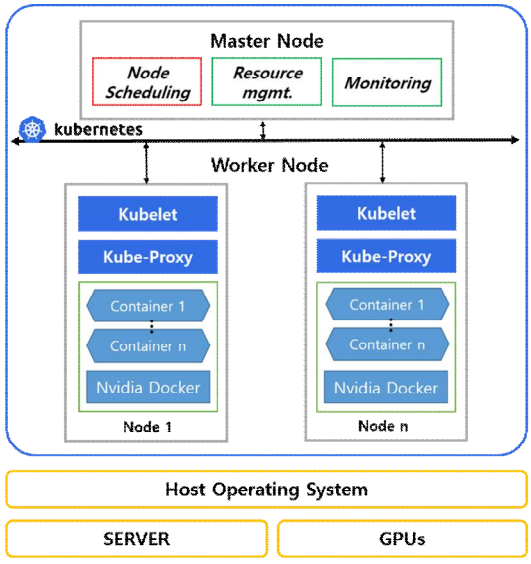


그림 2. GPU 공유 컨테이너 클러스터의 구조
Fig. 2. Architecture of container cluster with GPU sharing

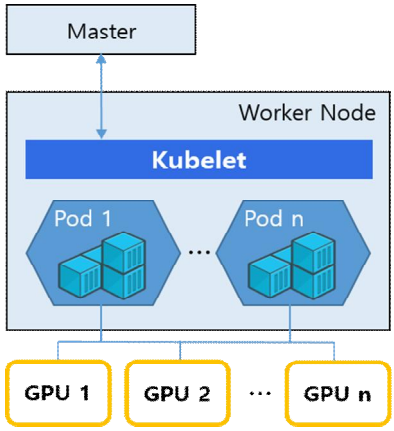


그림 3. 서버 간의 GPU 자원 공유
Fig. 3. Sharing GPU resources among multiple servers

노드가 전체 시스템을 제어한다. 노드 스케줄링 모듈은 작업 컨테이너에 적합한 노드를 선택하여 선택한 노드에 해당 작업을 할당한다. 또한, 노드 스케줄러는 물리적 GPU를 공유하기 위해 각 워커 노드에 대해 여러 개의 가상 서버(pod)를 만들고, 다중 응용의 실행 및 복제 등의 작업을 담당한다. 자원 관리 모듈은 각 노드의 가용 자원을 지속적으로 모니터링 하며, 단일 컨테이너의 상태 및 자원 소비량 모니터링을 지속한다.

워커 노드는 작업 컨테이너가 생성되고 실행되는 GPU 노드이다. 워커 노드는 마스터 노드의 Kubectl[11] 명령에 따라 가상 GPU, 응용 컨테이너 이미지 등을 수신하고 가상 서버를 실행한다. Kubelet[12]을 통해 마스터와 통신하고, 실행 중인 서버의 상태를 지속적으로 업데이트한다. Kube-proxy는 각 서버의 컨테이너에 대한 네트워크 프록시 또는 로드 밸런서를 담당한다. 본 시스템의 워커 노드는 여러 가상 서버를 실행하여 각 작업 컨테이너를 구성, 실행 및 관리한다. Nvidia Docker[13] 엔진은 모든 워커 노드에 설치되어 있다. 워커 노드는 마스터 노드로부터 Kubelet 에이전트를 통해 실행해야 하는 명령을 수신하며, 컨테이너 수행 결과와 상태를 마스터에게 전달한다.

쿠버네티스 API인 device plugin은 한 노드에 여러 서버를 구성하도록 수정하였다. 수정된 환경에서는 워커 노드의 Kubelet이 추가 정보 (GPU 메모리 용량과 같은 하드웨어 정보)와 함께 빌트인 정보를 수신한다. 이 정보를 바탕으로 각 GPU 노드에 대한 가상 서버를 구성한다. Pod은 하나 이상의 컨테이너로 구성되며, 각 컨테이너는 노드의 GPU에 할당되어 GPU를 공유한다. 수정된 device plugin에 의한 여러 서버의 GPU 자원 공유는 그림 3과 같다. 수정된 device plugin을 사용하여 GPU에서 응용이 동시 수행됨으로써 간섭이 발생할 수 있다. 본 프레임워크를 사용하여 응용의 동시 수행 시간을 측정하고, 응용이 단독으로 수행된 시간과 비교하여 동시 실행에 따라 발생할 수 있는 간섭을 측정할 수 있다.

Ⅳ. 결과 및 검증

1. 실험 환경

본 장에서는 제안된 프레임워크를 통해 multi-GPU 환경에서 컨테이너가 수행될 GPU 카드를 지정하여 실험을 진행하였다. 실험환경은 표 1과 같으며, 표 1에서 나타난 GPU 4개를 가진 하나의 노드로 실험환경을 구성하였다. 실험은 딥 러닝 및 과학 계산에 최적화된 GPU 가속 클라우드 플랫폼인 NVIDIA GPU Cloud(NGC) [10]에서 LAMMPS, GROMACS, Tensorflow, QMCPACK 컨테이너를 구동하여 실험하며 각 가상 서버에는 하나의 컨테이너가 할당된다. 컨테이너 구성은 NGC의 기본 구성을 따른다.

표 1. 실험 환경
Table 1. Experimental environment

	CPU	GPU
Architecture	Intel(R) Core(TM) i7- 5820K	Nvidia GeForce Titan Xp D5x
Core clock	3.30GHz	1.58GHz
Num of Cores	6 cores	3840 cores
Memory size	32GB	12GB
Threading API	-	Nvidia CUDA 10.1
Compiler	Gcc 5.4.0	Nvidia C Compiler (NVCC8.0)
OS	Ubuntu 16.04.3 LTS	Ubuntu 16.04.3 LTS

2. 수정된 Device plugin 평가

본 실험은 2개의 GROMACS 컨테이너를 기존 shared GPU device plugin을 사용하여 배치하였을 때 컨테이너가 수행될 GPU 카드가 임의로 선택됨을 보여준다. 기존 device plugin은 여러 GPU 중 컨테이너가 수행될 특정 GPU 카드를 지정할 수 없었으며, 사용할 카드의 수만 지정할 수 있었다. 그러므로 본 실험에서 각각의 컨테이너가 다수의 GPU 카드 중 하나만을 할당하도록 명세하여 30번의 실험을 진행하였으며 실험 결과는 표 2와 같다.

표 2. 기존 Device plugin을 사용한 GPU 카드 무작위 지정 결과
Table 2. Random assignment of GPU using existing shared device plugin

G2 / G1NGH	GPU0	GPU1	GPU2	GPU3
GPU0	3	5	2	5
GPU1	0	1	2	3
GPU2	0	0	0	5
GPU3	0	0	0	4

*G1 :GROMACS1, G2 : GROMACS2

원래의 shared device plugin을 사용하여 2개의 컨테이너를 배치하면 지정된 GPU가 매 실험마다 바뀌었다. 예를 들어 30회의 실험 중 GROMACS 컨테이너들은 각각 GPU0과 GPU0을 세 번, GPU0과 GPU1을 다섯 번, GPU 0과 GPU1을 2 번 선택하여 수행된 것을 확인할 수 있다. 이는 두 개의 컨테이너가 수행될 GPU가 임의로 선택됨을 보인다. 두 개의 컨테이너가 하나의 GPU를 공유하여 수행하면, 각각의 컨테이너는 1505초, 1507초 동안 수행된다. 하지만 각각 다른 GPU에서 수행하게 되면 각각 716초, 721초가 소요된다. 이를 통해 응용의

스케줄링 방식에 따라 실험 전체의 수행시간이 달라짐을 알 수 있다. 하지만 기존의 device plugin은 비결정적이기 때문에 실험 결과를 예측할 수 없으므로 본 논문에서는 수행될 GPU 카드를 지정할 수 있도록 device plugin을 수정하였다.

3. 정책에 따른 컨테이너 스케줄링 평가

표 3. HPC / DL 응용의 자원 사용량
Table 3. Resource usage of HPC / DL applications

Container	Memory requirement	Average GPU utilization
LAMMPS	8.3GB	23%
GROMACS	0.6GB	72%
Tensorflow	3.5GB	88%
QMCPACK	5.3GB	46.7%

2절의 실험을 통해 우리의 device plugin이 원하는 GPU 카드를 지정할 수 있음을 보여주었다. 이는 각 컨테이너를 스케줄링 정책에 따라 스케줄링이 가능함을 의미한다. 본 실험에서 사용한 스케줄링 정책은 다음과 같다 :

Load balancing : 부하의 균형을 맞추기 위해 서로 다른 자원을 요구하는 응용끼리 배치하여 서로 간섭하지 않고 수행되도록 한다.

Standalone : 같은 자원을 요구하는 응용끼리 그룹지어 배치한다.

Binpacking : 가용한 자원을 가진 노드에 최대 가능한 작업을 packing한다.

우리는 작업의 풀에 LAMMPS, GROMACS, QMCAPCK, Tensorflow 컨테이너가 각각 두 개씩 있다고 하였을 때, 위에서 설명한 세 개의 정책에 따른 스케줄링을 비교하였다. 각 응용의 메모리 요구량 및 계산 자원 사용을 나타내는 GPU utilization은 표 3과 같다. 그림 4는 응용의 특성 및 각 정책에 따라 배치된 컨테이너들을 설명한다. 각 하늘색 상자는 GPU 카드를 의미한다.

그림 4-(a)에서 Load balancing 정책은 서로 다른 컴퓨팅 자원 요구량을 가진 응용끼리 배치하기 위해서 GPU 커널이 실행되는 비율을 나타내는 평균 GPU utilization이 가장 큰 응용부터 배치하되, 각 GPU에 배치된 컨테이너의 utilization 합이 가장 작은 GPU를 선택한다. 평균 GPU utilization이 88%으로 가장 큰 Tensorflow 두 개 컨테이너와 71%인 GROMACS 두 개의 컨테이너가 각각 GPU 하나에 배치된다. 평균 utilization이 46.7%인 QMCPACK은 GROMACS가 배치되어 utilization 합이 가장 작은 GPU 카드로 배치된다. 마지막으로 23%의 a 평균 utilization을 갖는 LAMMPS 컨테이너는 Tensorflow가 배치된 GPU에 배치된다.

GPU0,1 : Tensorflow(88%) + LAMMPS(23%)
GPU2,3 : GROMACS(71%) + QMCPACK(46.7%)

그림 4-(b)에서 Standalone 정책은 같은 자원을 요구하는 응용인 같은 응용의 컨테이너가 같은 GPU 카드에 배치된다.

본 실험에서 각 GPU의 가용 메모리가 12GB이므로, 그림 4-(c)의 Binpacking 정책 작업의 풀에서 가장 메모리 요구량이 큰 컨테이너부터 배치하되, 응용의 GPU 요구량보다 가용 메모리 양이 많은 GPU 중 가장 가용 메모리 양이 적은 카드에 배치한다. 이는 8.3GB의 메모리 요구량을 가지는

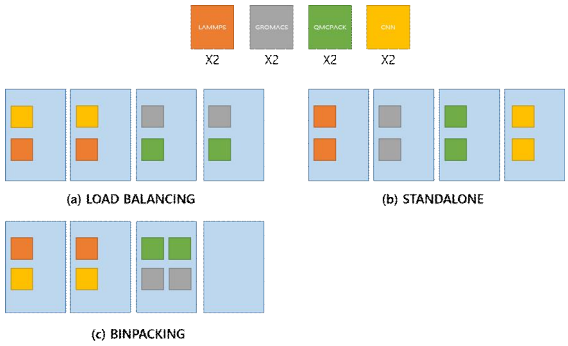


그림 4. 스케줄링 정책에 따른 컨테이너 배치
Fig. 4. Placement of containers according to different scheduling policies

LAMMPS가 같은 GPU에 배치되지 못하여 각각 다른 GPU에 배치된다. 5.3GB의 메모리 요구량을 가지는 QMCPACK 두 개의 컨테이너는 하나의 GPU로 배치되며, 각각 3.5GB의 요구량을 갖는 Tensorflow 컨테이너는 LAMMPS 컨테이너와 각각 같이 배치된다. 0.6GB의 요구량을 갖는 GROMACS 두 개 컨테이너는 QMCPACK 두 개의 컨테이너와 함께 배치된다.

GPU0,1 : LAMMPS(8.3GB) + Tensorflow(3.5GB)
GPU2 : QMCPACK(5.3GB) + QMCPACK(5.3GB) + GROMACS(0.6GB) + GROMACS(0.6GB)

실험의 결과는 표 4와 같다. 전체 워크로드에 각 응용이 두 개씩 존재하므로, 두 작업을 구분하기 위해 예를 들어 LAMMPS의 경우 L1, L2로 표현하였다. 전체 워크로드의 수행 시간은 Binpacking이 3450초, Standalone이 3067초, Loadbalance가 3111초로 Standalone이 가장 적은 시간이 소요된 것을 알 수 있다. 각 응용 별 수행 시간을 비교해보았을 때, LAMMPS는 Standalone일 때 3067, 3009초로 가장 좋은 성능을 보였다. GROMACS와 QMCPACK도 Standalone일 때 가장 수행시간이 작았는데, Tensorflow 컨테이너일 때는 Loadbalance 정책이 가장 적게 소요되었으며, Standalone이 가장 많이 소요된 것을 확인할 수 있었다.
서로 다른 자원을 사용하여 가장 적은 시간이 소요될 것 같았던 Loadbalance 정책이 같은 자원을 사용하는 Standalone 정책보다 더 오랜 시간이 소요된 것을 알 수 있다. 이는 응용의 전체적인 자원 사용을 보지 않고, 응용의 메모리 요구량과 평균 GPU utilization만 고려하였기 때문이다. 본 실험을 통해 수정된 device plugin으로 GPU를 지정하여 수행할 수 있어 각 스케줄링 policy에 따라 스케줄링이 가능하며, 스케줄링 방법에 따라 결과가 다름을 알 수 있다. 또한, 각 응용의 자원 사용을 고려할 때, 메모리 요구량과 평균 GPU utilization만 고려해서는 최적의 스케줄링을 할 수 없음을 보여준다.

표 4. 스케줄링 정책에 따른 컨테이너 배치 실험 결과
Table 4. Results of container placement experiment according to different scheduling policies

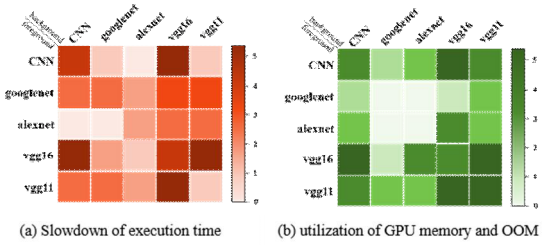
	L1	L2	G1	G2	T1	T2	Q1	Q2
Binpacking	3450	3449	2490	2824	85	85	703	1294
Standalone	3067	3009	2072	2089	147	148	854	802
Loadbalance	3105	3111	2185	2185	83	83	905	801

*unit : sec

**L : LAMMPS, G :GROMACS, T : Tensorflow, Q : QMCPACK

4. 동시 수행에 따른 응용의 간섭 측정

3절의 실험 결과를 통해 메모리 요구량과 평균 GPU utilization만 고려하여서는 최적의 스케줄링을 할 수 없음을 보였다. 이는 GPU에는 컴퓨팅 자원 외에도 여러 자원이 있어 함께 수행하는 각 응용이 여러 자원에 대하여 복잡한 자원 사용 패턴을 보이



(a) Slowdown of execution time (b) utilization of GPU memory and OOM

그림 5. 동시 수행에 따른 응용의 간섭
Fig. 5. A pair of applications interference (execution time slowdown on the left, GPU memory on the right)

며 서로 자원에 대해 경쟁하기 때문이다. 우리는 해당 프레임워크를 사용하여 공동 실행하는 응용 한 쌍에서 발생하는 용용의 쌍 별 간섭을 확인하기 위해 Tensorflow benchmark [14] 에서 ML 모델을 학습하는 5개의 Deep Learning (DL) 작업 (CNN, vgg16, vgg11, alexnet, googlenet)을 실행한다. 실행하는 응용을 imagetnet을 dataset으로 하여 image classification을 하는 응용이다. 그림 5는 서로 다른 작업이 함께 배치될 때 서로 다른 수준의 간섭이 발생함을 보여준다. 작업의 slowdown of execution, GPU 메모리에서 발생하는 간섭을 확인하였다.

그림 5 왼쪽은 한 쌍의 응용을 공동 실행 시 수행 시간을 각각 독립 시간에 비해 slowdown을 계산하여 간섭을 구하였다. 색상이 연할수록 낮은 간섭을 보이며, 색상이 진할수록 높은 간섭을 가짐을 나타낸다. 예를 들어, CNN-vgg11, vgg11-CNN 과 같이 같은 응용쌍의 실행의 경우 foreground, background에서 수행할 때 각각 서로 다른 간섭을 가지는 것을 확인할 수 있다. 또한, 그림 5의 오른쪽에서 보이는 것과 같이 OOM (Out of Memory)이 발생하는 CNN-vgg16, vgg16-CNN, vgg16-vgg11, vgg11-vgg16의 경우를 제외하고, 그림 5의 왼쪽과 비교하였을 때 간섭의 정도가 다름을 확인할 수 있다. 예를 들어, CNN과 vgg11을 동시에 수행하였을 때 메모리 활용도 측면에서는 비교적 높은 간섭이 일어나지만, 수행시간 측면에서는 slowdown이 많이 일어나지 않는다. 이는 응용 실행 시 발생할 수 있는 간섭이 공유되는 여러 자원으로부터 영향 받을 수 있음을 보여준다. 특히, vgg11의 경우 실행시간 측면에서 vgg11을 함께 돌렸을 때 간섭이 적게 나타났지만, CNN의 경우 높은 간섭이 있는 것을 확인할 수 있다. 3절의 실험에서

Standalone 방식으로 수행한다면, vgg11의 경우는 좋은 성능을 보이겠지만 CNN의 경우는 나쁜 성능을 보일 것이다. 이렇게 응용마다 동시 수행에 있어서 다른 특성을 보이며, 여러 자원의 경쟁으로 인한 간섭이 나타나기 때문에 본 논문의 GPU 공유 컨테이너 클러스터를 통해 간섭을 측정하는 것이 중요하다.

Ⅴ. 결 론

본 논문에서는 각 노드에 대해 가상 서버를 실행하여 GPU 공유를 가능하게 하는 프레임워크를 제안했다. GPU 메모리 및 컴퓨팅 리소스 등 GPU 자원을 많이 사용하지 않는 작업이라면, GPU 노드의 리소스 사용률이 저하될 수 있다. GPU에서 여러 응용이 동시 실행된다면, GPU 리소스 낭비 및 독점 문제가 해결된다. 제안한 GPU 공유 방법이 HPC 및 ML 워크로드를 대상으로 하여 스케줄링 정책(Binpacking, Standalone, Load balancing) 에 따라 쿠버네티스 프레임워크 위에서 작동할 수 있음을 실험을 통해 보였다. 실험 결과를 통해 GPU 메모리 및 컴퓨팅 리소스만을 고려해서는 최선의 스케줄링을 할 수 없음을 보이며, 이로 인해 동시 배치되는 응용끼리의 간섭을 해당 프레임워크를 사용하여 측정하였다.

마지막으로 본 논문을 확장하기 위한 향후 연구 방향을 다음과 같다. 개별 응용의 리소스 사용 특성을 연구하여 여러 자원이 간섭에 주는 영향을 확인하고, GPU에 함께 배치된 응용의 간섭을 고려한 스케줄링 정책을 연구할 예정이다.

References

[1] Amazon EC2, https://aws.amazon.com/ec2/

[2] Nimbix, https://www.nimbix.net/cloud-computing-nvidia

[3] Microsoft Azure, https://docs.microsoft.com/en-au/azure/virtual-machines/windows/sizes-gpu

[4] Alibaba, , https://www.alibabacloud.com/ko/product/gpu

[5] Google container, https://cloud.google.com/containers

[6] Kubernetes, https://kubernetes.io

[7] Fabiana Rossi, Matteo Nardelli and Valeria Cardellini. 2017. Horizontal and Vertical Scaling of Container-based Applications using Reinforcement Learning, In Cloud, 2019, IEEE International Conference on. IEEE, 329-339

[8] Gu, Jing, et al. "GaiaGPU: Sharing GPUs in Container Clouds." 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). IEEE, 2018.

[9] Song, Shengbo, et al. "Gaia Scheduler: A Kubernetes-Based Scheduler Framework." 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). IEEE, 2018.

[10] NVIDIA GPU Cloud, https://ngc.nvidia.com/

[11] kubectl, https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands

[12] kubelet, https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/

[13] NVIDIA Docker, https://github.com/NVIDIA/nvidia-docker/wiki/nvidia-docker

[14] Tensorflow CNN benchmarks, https://github.com/tensorflow/benchmarks/tree/master/scripts/tf_cnn_benchmarks

김 세 진 (Se-Jin Kim)

2019년 2월 : 숙명여자대학교 소프트웨어학부 컴퓨터과학전공 졸업

2019년 3월~현재 : 숙명여자대학교 컴퓨터과학과 석사과정

〈관심분야〉 클라우드 컴퓨팅, 고성능 컴퓨팅, 이기종 컴퓨팅

김 윤 희 (Yoon-Hee Kim)

1991년 숙명여자대학교 전산학과 졸업(학사).

1996년 Syracuse University 전산학과 졸업(석사).

2000년 Syracuse University 전산학과 졸업(박사).

1991년~ 1994년 한국전자통신연구원 연구원.

2000년~2001년 Rochester Institute of Technology 컴퓨터공학과 조교수.

2001년 ~ 2016년 숙명여자대학교 컴퓨터과학부 교수.

2017년 ~ 현재 숙명여자대학교 소프트웨어학부 교수.

<관심분야> 클라우드 컴퓨팅, 워크플로우 제어, 그리드/클라우드 관리