저작자표시-비영리-변경금지 2.0 대한민국

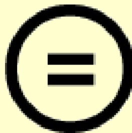
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학석사 학위논문

공유된 GPU 클러스터의 효율적인 자원 배분에 관한 연구

2020년 8월

서울대학교 대학원
컴퓨터공학부
서 장 호

공유된 GPU 클러스터의 효율적인 자원 배분에 관한 연구

지도교수 전 병 곤

이 논문을 공학석사 학위논문으로 제출함

2020년 6월

서울대학교 대학원

컴퓨터공학부

서 장 호

서장호의 석사 학위논문을 인준함

2020년 6월

위 원 장 염 헌 영 (인)

부위원장 전 병 곤 (인)

위 원 이 재 진 (인)

국문초록

딥 러닝 학습 작업은 상대적으로 비싼 계산 자원인 GPU를 적극적으로 활용하며, 이 때 여러 학습 작업이 공유하는 GPU 클러스터를 관리하는 자원 관리자를 도입하는 것이 일반적이다. 이 논문은 자원 관리자가 어느 작업에 주어진 자원을 재배정하거나, 그 양을 탄력적으로 줄이거나 늘릴 수 있는 역량을 갖추는 방법에 대해 탐구한다. 또한 이러한 방식을 시험하기 위해 제작한 시험적 구현체가, GPU들을 정적으로 분할하는 스케줄러에 비해 GPU 활용률과 전체 워크로드 처리에 걸린 시간 측면에서 우수한 성능을 보인 사례를 제시한다.

....

주요어: 자원 관리자, 스케줄러, 탄력적 자원 사용, 딥 러닝, 분산 학습, 자원 활용률, 응답 시간, 동적 자원 재배정

학 번: 2018-28102

목차

1. 개요 ·· 1
2. 배경 ·· 5
    2.1. 딥 러닝의 과정과 성능 ················ 5
    2.2. 자원 관리자 ·························· 7
    2.3. 딥 러닝을 위한 자원 관리자 ·········· 11
3. F-스케줄링 ································· 14
    3.1. 감속률과 공평성 ······················ 16
4. 시험적 구현체 ······························ 18
    4.1. 자원 모델링 ·························· 18
    4.2. 스케줄링 알고리즘 ···················· 19
    4.3. 시스템 아키텍처 ······················ 21
    4.4. 스케줄러 구현 ························ 22
5. 성능 평가 ·································· 25
    5.1. 실험 설정 ···························· 25
    5.2. 실험 결과 ···························· 27
6. 결론 ·· 30
참고문헌 ······································· 31
Abstract ······································· 34

표 목차

Table 1. 워크로드 ·· 26
Table 2. 워크로드 전체가 완료되기까지의 소요시간··················· 27

그림 목차

Figure 1. 어플리케이션의 상태·· 24
Figure 2. 모델에 대한 함수 f ·· 26
Figure 3. 워크로드에서 발생한 작업별 소요시간의 상세 분석 ········ 28
Figure 4. 워크로드 실행중 관측된 GPU 활용률의 누적분포함수 ····· 29

일람 목차

Listing 1. 실험적 구현체의 스케줄링 알고리즘을 표현한 의사코드 ·· 19

제 1절

개요

지난 수년에 걸쳐 기계학습, 그 중에서도 딥 러닝 기술은 빠르게 발전하여 이전에는 가능하지 않았던 수많은 부가가치를 생산하고 있다. 이에 발맞추어 딥 러닝 기술의 연구와 적용을 위한 자원의 수요도 증대되고 있으며, 오늘날의 대규모 계산 환경에서 딥 러닝은 이미 자원 사용의 큰 비중을 차지하고 있다. 그러나 딥 러닝은 GPU를 집중적으로 사용하는데, GPU가 포함된 계산 자원은 직접 구축하여도(on-premises), 클라우드를 이용하여도 GPU 없는 계산 자원에 비해 비용이 매우 비싸다. 예를 들어 아마존 웹 서비스 미국 동부 리전의 EC2 서비스에서 가상 CPU 코어 64개와 메모리 256GiB를 가진 m4.16xlarge 인스턴스는 시간당 3.20 USD가 과금되지만, CPU 코어와 메모리는 더 적게 제공하나 NVIDIA V100 Tensor Core GPU가 탑재된 p3.8xlarge 인스턴스는 시간당 12.24 USD가 과금된다. [1]

이에 따라, GPU 자원의 공유를 가능하게 하는 자원 관리자(resource manager)의 효율이 중요한 문제로 대두되고 있다. 자원 관리자는 클러스터의 여러 자원 상태를 추적하고, 사용자로부터 여러 작업(job)을 의뢰받아, 클러스터 전반에 걸친 높은 자원 활용률(resource utilization)과, 작업간의 공평성(fairness)을 목표로 작업을 자원에 적절히 배치하는 스케줄링을 담당한다.

딥 러닝 학습 작업을 스케줄링 하는 것은, 갱(gang) 스케줄링과 지역성(locality) 요구사항으로 인해 작업의 대기 시간(queuing delay)이 길어지

고 클러스터의 활용률(utilization)이 낮아지는 어려움을 겪기 쉽다. 작업을 구성할 때 적용된 하이퍼파라미터(hyperparameter)가 특정 GPU 개수에 맞추어져 있기 때문에, 정해진 개수의 GPU를 모두 사용할 수 있을 때에만 진행되는 갱 스케줄링을 요구한다. 또한, GPU 자원이 충분히 남지만, 이들 GPU가 여러 머신에 흩어져 있어 자원의 지역성이 열악하다면, 지역성이 우수한 자원이 사용 가능해질 때까지 작업이 대기하는 경우가 생길 수 있다. 이러한 경우 작업이 시작되지 못하고 기다리는 시간이 길어지며, 잉여 자원은 사용되지 않는 채로 남게 된다. [2]

자원 관리자가 진행 중인 작업에 주어진 작업을 동적으로 재배정하는 것이 가능해진다면 이러한 한계를 극복하는데 도움이 됨은 선행 연구를 통해 알려져 있다. [3] 즉 기존 작업을 적극적으로 재배치해 요구 자원을 확보한 후 대기 중인 작업을 시작하거나, 일단 열악한 자원에 작업을 배치한 후 이후 상황에 따라 우수한 자원으로 재배정하는 등의 전략이 가능해진다.

한편, GPU 사용을 주된 사용례로 가정하지 않은 전통적인 자원 관리자에서는, 작업이 적절한 자원 요구사항을 도출해 자원 관리자에게 제공하거나 [4], 자원 관리자가 자원의 일부를 작업에게 제시[5]하는 정보의 교환이 발생했다. 이 때, 자원을 작업과 자원 관리자가 공통으로 이해할 수 있는 사양으로 기술하기 위해 추상화하는 과정에서, 자원간의 위계 정보가 무시되는 등의 정보 손실이 발생하나[6], 이러한 규약은 대규모 데이터 분석 작업을 위한 스케줄러를 작성하는데 널리 사용되었다. 그러나 GPU를 주로 사용하는 딥 러닝 학습 과정은 주어진 자원들 사이의 지역성(locality), 인접한 작업간의 간섭, 자원간의 위계 구조(topology) 등이 성능에 큰 영향을 미치는 경우가 많다. [3] 그러므로 기존의 자원 관리자

가 상정하는 방식대로 작업과 자원 관리자간의 정보 교환이 발생할 때, 스케줄링의 품질을 높이는 데 사용될 수 있는 중요한 정보가 손실된다고 생각할 수 있다.

자원 관리자가 작업에 주어진 자원을 동적으로 재배정하면서, 자원의 세부 사양이 성능에 미치는 영향을 충분히 고려할 수 있으려면, 주어진 자원에 대해 예상 성능을 계산하는 함수가 필요하다고 이 논문은 주장한다. 이러한 함수를 산술적으로 온전하게 기술하는 것은 현실적이지 않으나, 기존에 유사한 작업을 수행한 과거 기록이나, 상정한 자원을 실제로 제공하고 학습 과정의 몇 스텝(step)을 시험 삼아 수행해보는 것으로 함수의 개형을 귀납하는 것이 가능하다.

이 논문의 기여사항은 다음과 같다.

이 논문은 작업에 주어진 자원을 재배정하거나, 그 양을 탄력적으로 줄이거나 늘릴 수 있는 자원 관리자의 설계를 제안한다. 이 때 주어진 자원에 대한 예상 성능을 반환하는 함수는 주어져 있다고 가정한다. 자원 관리자는 이 함수의 내부를 이해할 필요는 없지만, 자신이 제공할 수 있는 자원의 다양한 부분집합에 대한 예상 성능을 이 함수를 통해 얻을 수 있다. 이 때 자원 관리자는 여러 작업에게서 받은 함수를 바탕으로, 클러스터 전반의 처리율(throughput)을 높이려고 시도한다. 자원 관리자는 이미 실행중인 작업에 자원을 추가하거나, 작업이 사용하고 있는 자원의 일부만을 빼앗을 수 있다. 새로운 작업이 의뢰되거나, 기존의 작업이 완료되어 자원을 요구하지 않게 되는 경우 자원 관리자는 주어진 함수를 바탕으로 새로운 상황에서의 배치를 다시 계산하여, 기존 작업들이 점유하는 자원을 재조정한다.

또한 이 논문은 각 작업이 제공한 함수를 바탕으로 작업 간의 공평성을 계산할 수 있는 측정 방식으로 감속률의 분산을 제안한다. 감속률이란 어느 작업이 주어진 자원을 이용하여 실행될 때의 성능이, 클러스터 전체의 자원을 독점했을 때 보이는 성능에 비해 얼마의 비율로 느린지의 값이다. 이 비율은 작업이 자원을 타 작업과 공유한다는 이유로 성능상의 손해를 얼마나 보는지의 값으로 이해할 수 있다. 이 때, 자원 관리자는 작업의 감속률의 분산이 일정 수치 미만이 되는 한도 내에서 클러스터 전반의 처리율이 가능한 높아지도록 자원을 배치한다.

제 2절

배경

2.1. 딥 러닝의 과정과 성능

딥 러닝의 학습(training) 단계는, 주로 인공신경망을 모사한 모델을 상정하고, 모델을 구성하는 파라미터의 적절한 값을 데이터를 통해 찾아나가는 과정이다. 이렇게 완성된 모델에 데이터를 입력하면 예측값을 얻을 수 있는데, 모델의 종류와 사용한 데이터에 따라 이 예측값은 이미지 분류, 자연어 인식, 번역과 같은 다양한 용도의 작업에 활용될 수 있다. 딥 러닝의 학습 과정은 대량의 데이터를 활용하여 이 예측값의 품질을 높이는 과정이라고 할 수 있다. 일반적으로 학습 과정은 여러 번의 연속된 스텝(step)으로 구성된다. 한 스텝에서, 데이터의 부분집합을 입력으로 삼아 현재 모델의 예측값을 계산한다. 그리고 이 예측값이 바람직한 예측값과 얼마나 차이가 나는지를 보고, 이 차이를 피드백으로 사용하여 모델 내의 파라미터를 조정한다. 이러한 스텝이 반복되어, 모델의 예측값과 바람직한 예측값의 차이가 일정 이하로 줄어들면, 일정 이상의 품질을 확보한 모델을 얻은 것이다.

학습 과정을 통해 모델이 일정 이상의 품질을 보여주기까지 얼마나 걸릴지 예측하는 것은 쉽지 않다. 모델의 예측값과 바람직한 예측값의 차이가 줄어드는 추세를 보고 예측하려고 해도, 모든 딥 러닝 모델의 학습 과정이 이 차이가 예측 가능한 속도로 줄어드는 결과를 보이는 것은 아니다. 이 때 딥 러닝 학습을 의뢰한 사용자는 수행할 스텝의 최대 횟수를 지정하여, 이 횟수만큼 스텝이 실행되면 학습 과정이 종료되도록 한

다. 궁극적으로 딥 러닝의 학습 과정에서 성능이란, 일정 이상의 품질을 가진 모델을 확보하는데 걸리는 시간이 얼마나 짧은지가 되겠지만, 이 논문에서는 처리율(throughput), 즉 단위시간당 얼마나 많은 스텝을 수행할 수 있는지를 성능 지표로 삼기로 한다.

그러나 단위시간당 수행할 수 있는 스텝의 수 역시 일반화하여 예측하기는 어려운데, 이는 이것은 딥 러닝을 통해 풀고자 하는 문제에 따라 모델의 내부 구조가 다양하며, 한 스텝의 학습에 사용할 데이터 크기(batch size)나 러닝 레이트(learning rate)가 성능에 영향을 미치고, 학습 과정을 수행하는 환경 역시 다양하기 때문이다. 특히나 이는 여러 GPU, 또는 여러 머신을 동원하여 학습을 수행하는 분산 딥 러닝(distributed deep learning) 에서 특히 그러하다. 예를 들어, ResNet-50과 같은 모델은 분산 작업을 수행하는 GPU가 여러 머신에 흩어져 있을 때 성능이 크게 감소하는 반면, InceptionV3과 같은 모델은 상대적으로 이와 같은 자원의 지역성(locality)에 영향을 덜 받는 모습을 보인다. 분산 작업에 사용된 GPU가 같은 머신에 부착되어 있는지, 같은 머신이라면 같은 PCIe 스위치를 공유하는지, GPU들이 다른 머신에 있다면 이 머신간의 통신 대역폭은 얼마인지 등, 학습 과정의 성능에 영향을 미칠 수 있는 요인은 다양하다. [3]

한편으로 딥 러닝 학습 과정은 매 스텝마다 같은 구조의 연산을 수행하므로, 단위시간당 완료되는 스텝의 수는 자원 환경이 변하지 않는다면 일정하다는 특성 또한 보인다. [3] 따라서 몇 스텝을 시험 삼아 수행해봄으로서 이후에 작업이 보일 성능을 예측하는 것이 가능하다.

2.2. 자원 관리자

클러스터 자원의 활용률을 높이고 각 자원간의 공평성을 보장하기 위한 연구 및 개발은 딥 러닝이 널리 쓰이기 이전부터 진행되어 왔다. 또한 이러한 자원 관리자를 연구, 개발 및 사용한 경험이 오늘날 GPU 자원 관리자의 설계에 영향을 미치기도 한다. 이 단락에서는 잘 알려진 두 자원 관리자에 대해 다룬다.

2.2.1. YARN

YARN[4]은 Apache Hadoop 2.0[7] 설계의 핵심이 되는 구성요소로, 자원 관리를 담당하는 리소스 매니저(Resource Manager)와 작업 모니터링을 담당하는 어플리케이션 마스터(Application Master)를 분할한 것이 특징이다. 리소스 매니저는 클러스터를 구성하는 각 노드에 하나씩 존재하는 노드 매니저(Node Manager)들이 제공한 정보로부터 클러스터 전체가 가진 리소스의 양과 상태를 추적한다. 사용자가 리소스 매니저에 특정 어플리케이션을 띄워 달라는 요청을 하면, 해당 어플리케이션의 로직을 수행하는 어플리케이션 마스터가 생성되어 실행된다. 어플리케이션 마스터는 어플리케이션 로직에 따라, 분산 작업에 필요한 컨테이너를 리소스 매니저에게 요청한다.

어플리케이션 마스터가 컨테이너 할당을 요청할 때 리소스 매니저에게 제공하는 정보는, 컨테이너에 필요한 자원의 사양이다. 예를 들어 CPU 코어 몇 개가 필요한지, 호스트 메모리의 양을 얼마나 필요로 하는지 등이다. 또한 선택적으로 리소스 제약사항(constraint)을 둘 수 있는데, 예를 들어 특정 라벨이 달려있는 노드에 컨테이너를 할당하기를 요구할 수 있다. 이와 같은 자원 요청 규약은, 대부분의 클러스터 환경이 수용할 수 있을 정도로 충분히 일반적이면서, 대규모 데이터 처리 시스템을 수행하

는 컨테이너를 할당하는데 필요한 수준의 정보를 제공한다.

그러나 이 규약은 어플리케이션 마스터가 자신이 요구하는 자원의 사양을 하나의 구체적인 값으로 정해서 표현하도록 강제하기 때문에 유연함이 떨어진다. 클러스터에 유휴 자원이 부족함에도 불구하고 어플리케이션 마스터가 강력한 사양의 컨테이너를 요청하면 큐 지연(queuing delay)이 길어지거나 할당에 실패하는 경우가 발생하고, 클러스터에 유휴 자원이 풍부함에도 필요 최소한의 사양만 가진 컨테이너를 요청하면 클러스터 전체의 자원 활용률이 떨어지는 상황을 초래한다.

또한 YARN의 리소스 요청 규약은 다양한 클러스터 환경에서 사용될 수 있도록 일반화되어 있기 때문에, 요구되는 자원의 계층 구성(topology)을 지정하는 것 역시 노드 수준 또는 랙 수준의 지역성 요구사항(locality constraint)을 지정하는 것으로 한정되어 있다. [8] 같은 노드를 공유하는 GPU 자원이라 할지라도 PCIe 스위치를 공유하느냐에 따라 성능 차이가 발생할 수 있고, 같은 랙에 존재하는 노드들이라 할지라도 스위치 구성에 따라 노드간의 대역폭이 차이가 날 수 있음을 생각해보면, 이 규약은 자원의 지역성이나 계층 구성에 따라 성능이 민감하게 변화하는 모델이 충분한 성능을 내도록 돕기에는 한계가 있다.

위와 같은 문제는 사용 가능한 자원이 무엇인지, 그 자원간의 계층관계나 세부사항은 어떠한지 어플리케이션 마스터가 알 수 없기 때문에 발생한다. 또한 어플리케이션 로직이 어플리케이션 마스터 내에 감추어져 있기 때문에, 리소스 매니저는 어플리케이션이 보이는 성능 특성이나 자원의 지역성에 대한 민감도 등을 추론하기 힘들다.

2.2.2. Mesos

Apache Mesos[5]는 컴퓨터 클러스터를 관리하고, 여러 스케줄러나 프레임워크를 그 위에 올려 사용할 수 있는 통합 자원 관리 레이어를 목표로 설계 및 개발된 프로젝트이다. 메소스 마스터(Mesos Master)는 각 노드에 존재하는 메소스 에이전트(Mesos Agent)가 보고한 정보를 바탕으로, 클러스터 전체가 가진 리소스의 양과 상태를 추적한다. 메소스 마스터는 또한 Hadoop 스케줄러, MPI 스케줄러와 같은 여러 스케줄링 프레임워크와 통신하는데, 클러스터에 유휴 자원이 있는 경우 스케줄러에게 이 자원을 사용할 것인지를 제안(resource offer)한다. 스케줄러는 제안 받은 자원이 유용하리라고 판단되면 이 자원을 이용하여 태스크(task)를 실행하라는 응답을 메소스 마스터에게 보내고, 메소스 마스터는 해당 메소스 에이전트에게 명령하여 태스크가 실행되도록 한다.

메소스 마스터가 스케줄러에게 자원을 제안할 때 제공하는 정보는, 해당 자원의 구체적인 사양과 어느 노드에 위치한 자원인지이다. 예를 들어 어느 노드에 CPU 코어 몇 개를 이용할 수 있으며, 메모리의 양은 얼마가 사용 가능한지 등이다. 스케줄러는 이를 검토하여 자신이 중요하게 여기는 리소스 제약사항에 미달하는 자원은 거절함으로서 제약사항을 실현한다. 이와 같은 자원 제안 규약은, 개별 스케줄러의 특성에 달린 리소스 제약을 자유롭게 구현할 수 있을 정도로 충분히 강력하면서, 이 때 메소스 마스터가 각 스케줄러가 실행하는 어플리케이션의 특성을 알지 못하더라도 괜찮다는 장점이 있다.

그러나 이 규약은 자원 제안에 대한 스케줄러의 응답이 본질적으로 수용 또는 거절이라는 점에서 한계가 있다. 또한 스케줄러는 지금 제시된 자원 제안을 수용 또는 거절하고 나면, 이후에 또 어떤 제안이 오게 될 지

알 수 없다. 따라서 앞으로의 자원 상황에 대해 낙관하는 스케줄러는 다소 부족한 사양의 자원 제안을 거절하고 충분한 사양을 갖춘 제안이 만들어질 때까지 기다릴 것이다. 이후 시간이 지나도 유휴 자원이 충분해지지 않으면, 불필요하게 시간을 낭비한 결과를 초래한다. 반대로 앞으로의 자원 상황을 비관적으로 보는 스케줄러는, 다소 부족한 사양의 자원이라도 적극적으로 받아들이려 할 것이다. 이후 시간이 지남에 따라 유휴 자원이 충분해진다면, 충분한 사양의 제안이 오기를 기다리는 게 작업의 실행 측면에서 나은 선택이었을 수 있다.

또한 Mesos의 자원 제안은 유휴 자원이 발생한다면 이를 제안하는 것이 기본이기 때문에, 스케줄러가 실행하는 어플리케이션의 성능 특질을 적극적으로 고려하여 자원을 재배치해주기는 힘들다. 예를 들어, 자원 지역성에 대해 비교적 둔감한 작업이 지역성이 우수한 자원을 점유하여 실행되고 있고, 이 상태에서 자원 지역성에 민감한 작업을 수행해야 한다면, 기존에 수행중인 작업을 지역성이 떨어지는 자원으로 재배정하고 기존 자원을 새 작업에 제안하는 것이 좋은 선택일 수 있다. 유휴 자원을 스케줄러에게 제안하는 규약으로는 이러한 적극적인 자원 재배정을 구현하는 것이 쉽지 않다.

위와 같은 문제는 각 스케줄러가 실행하는 어플리케이션의 성능 특성을 메소스 마스터가 알 수 없기 때문에 발생한다. 메소스 마스터가 다양한 스케줄러를 지원할 수 있도록 자원 제안이라는 일반화된 개념을 도입한 것이 Mesos의 설계 사상임을 고려하면, 이는 Mesos가 상정한 주된 사용 사례를 감안한 결과라고 보아야 할 것이다.

2.3. 딥 러닝을 위한 자원 관리자

기존 대규모 데이터 처리를 위한 자원 관리자의 한계를 극복하기 위해, 딥 러닝 어플리케이션의 특이성을 잘 활용하기 위한 연구를 바탕으로 딥 러닝을 위한 자원 관리자가 등장했다. 이러한 자원 관리자는 학습되고 있는 딥 러닝 모델의 성능 특성에 대해 이미 알려진 지식이 있다고 가정하거나, 딥 러닝 프레임워크와의 연동을 통해 보다 효율적인 자원 관리를 추구한다.

PyTorch[9], TensorFlow[10]와 같은 딥 러닝 프레임워크가 널리 사용되고 있기 때문에, 딥 러닝 프레임워크와 자원 관리자간의 연동이 강화되면 해당 프레임워크 위에서 실행되는 딥 러닝 학습 작업이 이로 인한 효과를 볼 수 있게 된다. 딥 러닝 프레임워크는 모델의 구조, 각 스텝의 진행상황, 각 스텝에서 모델이 내놓은 예측치와 바람직한 예측치의 차이 등 어플리케이션에 특이적인 속성을 잘 이해한다. 따라서 딥 러닝 프레임워크를 수정할 수 있다면, 개별 딥 러닝 작업을 하나하나 수정하는 것에 비해, 어플리케이션 수준의 정보를 자원 관리자가 이용할 수 있도록 연동하는 것을 더 쉽게 달성할 수 있게 된다.

2.3.1. Gandiva

Gandiva[3]는 딥 러닝 학습이 여러 스텝으로 이루어져 있으며, 각 스텝이 보이는 성능상의 특질은 일정하다는 점을 이용하는 스케줄링 프레임워크이다. Gandiva는 다양한 자원 환경에서 학습 스텝이 보이는 성능 특성을 알아내기 위해 작업을 다양한 자원 환경에 노출시킨다. 하나의 GPU에 여러 학습 작업을 동시에 실행시켰을 때 작업간의 간섭이 성능에 미치는 영향을 알아보기도 하고 (packing), 주어진 GPU의 개수를 늘리거나 줄여보기도 하며(grow-shrink), 작업을 다른 GPU로 이전하기도

한다 (migration). 이를 통해 자원의 지역성이나 작업 간의 간섭이 특정 모델의 학습 성능에 미치는 영향을 파악하면, 이러한 특성이 앞으로의 스텝이 수행될 때에도 일정하게 유지될 것이라는 가정을 한다.

Gandiva는 자원 관리자가 딥 러닝 학습 작업의 성능 특성을 파악할 수 있도록 해준다는 중요한 개선점이 있다. 자원 관리자는 자신이 관리하는 자원의 현황과 특성을 가장 잘 파악할 수 있으며, 이제 각 작업의 성능 특성도 알 수 있으므로 이 두 정보를 조합하여 효율적인 스케줄링이 가능하다. 그러나 이러한 성능 특성을 작업 실행 중에 여러 설정을 시도해봄으로써 파악해야 한다는 한계점이 있다.

2.3.2 Tiresias

Tiresias[11]는 작업의 평균 완료 시간을 단축시키고 GPU 자원의 활용률을 높이기 위해 작업들의 완료 시간의 분포를 활용하는 자원 관리자이다. Tiresias는 실행 가능한 여러 작업 중 가까운 미래에 완료될 가능성이 가장 높은 작업에게 우선권을 주는 스케줄링을 통해 작업의 평균 완료 시간을 단축시킨다. 만약 작업 완료 시간의 분포가 주어지지 않는다면, 지금까지 제공받은 GPU 시간의 합이 가장 적은 작업을 우선한다.

또한, Tiresias는 작업을 스케줄링할 때 작업에 주어진 모델을 프로파일링하여 모델을 구성하는 텐서 크기의 비대칭도(skew)가 심한지 확인한다. 크기가 큰 텐서가 분산 수행에서 네트워크 사용량의 불균형을 초래하므로, 텐서 크기의 비대칭도가 심한 모델은 지역성이 우수한 자원에서 학습시키고, 그렇지 않은 모델은 지역성과 관련한 자원 조건을 가하지 않는다.

Tiresias는 특정 작업의 수행 시간에 대한 정확한 예상치를 요구하지 않고도, 가장 짧은 시간 내에 끝나는 작업을 우선하는(shortest remaining time first) 스케줄링을 구현한다는 장점이 있다. 그러나 어느 작업이 완료될 때 까지 걸리는 시간은 주어진 자원에 따라 달라질 수 있으므로, 각 작업에 할당될 자원이 제시되지 않은 상황에서 작업 완료 시간의 분포를 얻어낼 수 있는 방법이 분명하지 않다.

제 3절

F-스케줄링

F-스케줄링이란 주어진 자원을 활용하여 달성할 수 있는 최선의 처리율을 반환하는 함수가 있다는 가정 하에, 이를 활용하여 작업을 자원에 스케줄링하는 규약이다. 이 규약 하에서, 자원 관리자는 자신이 관리하는 자원을 분배했을 때 작업들이 어떠한 성능을 보일지 함수를 통해 예측할 수 있다. 자원 관리자는 자원의 상태와 계층구조를 잘 이해하고 있고, 또한 작업들의 성능 특성을 함수를 통해 예측할 수 있으므로, 두 유형의 정보를 조합하여 효율적인 스케줄링을 할 수 있다.

어떤 딥 러닝 작업 M에 대해 자원 관리자에게 제시되는 함수는 아래와 같다.

$$f_M(R) = T$$

이 때 작업 M에게 자원 R이 주어지면, 이 때 R을 활용하여 단위시간당 처리할 수 있는 최대의 스텝의 수가 T인 것으로 본다. M은 주어진 GPU의 사양, 수, 지역성, 주어진 노드의 사양과 수, 노드 간의 통신 대역폭 등을 포함할 수 있다.

위의 f와 같은 함수를 통해 각 작업의 성능 특성이 표현될 수 있다. 예를 들어 작업에 동원되는 GPU들의 지역성에 따라 성능이 크게 변하는 작업은 f의 입력에서 GPU들의 지역성을 변화시킬 때 결과값 T가 크게 변할 것이다. GPU의 지역성에 대해 둔감한 작업은 같은 조건에서 T의

변화가 크지 않을 것이다.

자원 관리자는 함수 f의 내부 구조를 이해할 필요는 없지만, 스케줄링의 후보가 되는 자원을 f의 입력값으로 줌으로서 스케줄링 결정에 도움이 되는 정보를 획득할 수 있다.

함수 f를 통해 표현된 성능 특성은, 해당 특성이 나타나게 하는 자원이 클러스터에 존재하지 않는다면 자연스럽게 무시된다. 예를 들어, 여러 노드를 점유하여 실행할 때 성능이 크게 우수해지는 작업은, 여러 노드를 포함하는 자원 M에 대해 주어지는 처리율 T가 커지도록 함수 f를 구성할 수 있다. 이는 이 작업이 여러 노드를 통해 분산 수행되는 것을 강력하게 선호한다는 뜻이기도 하다. 이 선호를 자원 제약사항(resource constraint)를 통해 표현한다면, 애초에 하나의 노드만 관리하는 자원 관리자에서는 스케줄링이 불가능해지는 결과로 이어진다. 반면 F-스케줄링에서 자원 관리자가 하나의 노드만 관리하고 있다면, 여러 노드를 점유하여 작업을 실행하는 경우는 애초에 f의 입력으로 주어지지 않을 것이다. 분산 작업시 성능이 우수해진다는 작업의 특성은 무시되지만, 자원 관리자가 자신이 관리하는 자원을 동원하여 최선의 스케줄링을 시도할 수 있게 해준다.

이미 일정한 자원을 점유하여 실행중인 작업에 대해서도, 해당 작업에 관해 주어진 함수 f는 자원 관리자가 알고 있기 때문에, 작업에 주어진 자원에 변화를 주었을 때 어떤 성능이 나타날지 예측할 수 있다. 예를 들어 여러 작업이 수행중이고, 모든 자원이 어느 작업에 할당되어 포화상태인 클러스터에 새로운 작업이 추가된다면, 자원 관리자는 자원의 일부를 반환하여도 가장 성능 하락이 적은 작업으로부터 우선적으로 자원

을 반환받을 수 있다.

3.1. 감속률과 공평성

다양한 규모와 성능 특성을 가지는 작업에 대해 작업 간 공평성을 논하려면, 공평하다는 것이 무엇인지, 그리고 공평함의 정도를 어떻게 수치로 표현할 것인지에 대해 생각해야 한다. [12]

F-스케줄링에서 함수 f가 내놓는 값이 처리율이므로, 각 작업간의 처리율의 격차가 최소가 되도록 하는 것이 공평이라고 생각할 수 있다. 이 경우 각 작업이 보이는 처리율의 분산이 불공평함의 정도가 될 것이다. 그러나 이러한 방식은 실제로는 공평하게 작동하지 않은데, 같은 자원을 할당한다 하여도 계산의 규모가 크고 복잡한 모델을 학습하는 작업은 작고 간단한 모델을 학습하는 작업에 비해 처리율이 낮을 수 있기 때문이다. 이 때 작업 간의 처리율의 격차를 줄이려고 하면 크고 복잡한 모델을 학습하는 작업이 그렇지 않은 작업에 비해 불공평하게 우대받게 된다.

따라서 F-스케줄링에서 공평성이란 작업간의 감속률의 격차를 줄이는 것으로 정의한다. 어느 작업 M에게 자원 R을 할당했을 때, 감속률 S_M은 다음과 같이 정의된다.

$$S_M(R) = \frac{f_M(R)}{f_M(U)}$$

여기서 U는 클러스터가 보유한 모든 자원의 합으로 정의된다. 따라서 감속률이란, 자원 관리자가 동원할 수 있는 모든 자원을 동원하여 나타난

최선의 처리율에 비해서, 어느 자원 R을 할당받았을 때의 최선의 처리율의 비이다. 작업이 지닌 계산의 규모나 복잡성은 주어진 자원이 달라진다 하여 변하지 않으므로, 감속률을 통해 공평성을 측정할 때 계산의 규모나 복잡성에 의한 효과는 상쇄된다.

이 때 클러스터에서 실행중인 여러 작업에 대해 감속률의 분산이 불공평함의 지표가 된다. 따라서 자원 관리자는 감속률의 분산이 일정 수치 이하가 되도록 유지하는 선에서, 개별 작업의 처리율의 합이 최대가 되도록 자원을 할당한다.

제 4절

시험적 구현체

이 단락에서는 F-스케줄링을 시험적으로 실증하기 위한 구현체에 대해 설명한다.

4.1. 자원 모델링

이 시험적 구현체에서는 함수 f의 입력으로 주어지는 자원에 대한 모델링이 크게 단순화되어 있다. 함수 f는 작업을 여러 노드에 걸쳐 분산하여 실행하는 경우는 고려하지 않으며, 단일 노드에 동일한 사양의 GPU가 여러 개 있을 때 이 중 몇 개를 할당받아 실행하는지만 고려한다. 즉 한 노드에 여러 작업이 동시에 실행될 때 발생하는 간섭 효과에 대해서도 고려하지 않는다.

따라서 모델 M을 학습하는 작업에 대한 함수 f는 다음과 같다.

$$f_M(N) = T$$

여기서 N이 할당된 GPU의 개수이며, T는 주어진 GPU를 모두 활용하여 최대한으로 달성할 수 있는 단위시간당 스텝 수이다. 만약 GPU 메모리 부족 등으로 N개의 GPU를 활용하여 학습 진행이 불가능한 경우, T는 0이다.

4.2. 스케줄링 알고리즘

스케줄링 알고리즘은 작업의 감속률의 분산을 v_{bound} 미만으로 유지하면서, 작업들의 처리율이 최대가 되도록 하는 자원 할당을 찾는다. 이 때, 모든 가능한 할당의 경우의 수에 대해 감속률의 분산과 처리율의 합을 구하는 알고리즘은 대규모 클러스터에서 현실적이지 않은 시간 복잡도를 보이므로, 탐욕적 알고리즘(greedy algorithm)을 통해 이를 근사한다.

주어진 작업을 담은 배열이 A이고, 감속률의 분산의 상한을 v_{bound}, 사용 가능한 GPU의 개수를 N으로 두면, 알고리즘은 Listing 1 과 같다. 이 알고리즘은 주어진 작업이 요구하는 최소한의 GPU를 할당해 둔 다음, 여분의 GPU 각각에 대해 작업을 하나 골라 추가하는 방식으로 작동한다. 이 때 감속률의 분산이 지정된 수치 미만이 되도록 작업을 골라 GPU를 추가하려고 하며, 그러할 방법이 없다면 감속률의 분산이 그나마 낮은 방법을 선택한다. 그러한 방법이 둘 이상이라면, 처리율의 합이 가장 높은 방법을 선택한다.

```
def scheduler(A, v_bound, N):
    n  := N
    job_to_num_gpus := {}
    for M in A:
        num_min_gpus = get_min_gpus(M)
        if num_min_gpus > n:
            continue
        job_to_num_gpus[M] = num_min_gpus
        n -= num_min_gpus
```

```
    if len(job_to_num_gpus) == 0:
        return
    while n > 0:
        plan_candidates = []
        for M in job_to_num_gpus.keys():
            plan_candidate = job_to_num_gpus.copy()
            plan_candidate[M] += 1
            plan_candidates.append(plan_candidate)
        job_to_num_gpus = pick_plan(plan_candidates, v_bound)
        n -= 1
    for M in A:
        if M in job_to_num_gpus:
            schedule(M, job_to_num_gpus[M])


def pick_plan(plan_candidates, v_bound):
    fair_plans = []
    unfair_plans = []
    for plan_candidate in plan_candidates:
        slowdown_variance = \
             calculate_slowdown_variance(plan_candidate)
        if slowdown_variance < v_bound:
            fair_plans.append(plan_candidate)
        else:
            unfair_plans.append(plan_candidate)
    if len(fair_plans) > 0:
```

```
        return sorted(fair_plans, key=lambda p: \
                calculate_throughput_sum(p), reverse=True)[0]
    else:
        return sorted(unfair_plans, key=lambda p: \
                calculate_slowdown_variance(p))[0]
```

Listing 1: 실험적 구현체의 스케줄링 알고리즘을 표현한 의사코드

4.3. 시스템 아키텍처

구현체는 클러스터 전체를 통틀어 하나 존재하는 스케줄러와, 실행된 각 어플리케이션마다 존재하는 어플리케이션 마스터, 그리고 작업을 실행하는 워커로 구성된다. 각 구성요소의 역할과 책임은 다음과 같다.

- **스케줄러**는 클러스터에서 사용 가능한 자원과 그 상태를 추적하며, 어플리케이션의 상태를 기록한다. 새로운 어플리케이션이 실행되거나, 기존 어플리케이션이 종료되는 등의 사건에 반응하여 스케줄링 알고리즘을 작동시키고, 현재 시행되고 있는 자원 배정보다 나은 배정이 있다고 판단되면 관련된 어플리케이션에 명령을 보내 자원을 재조정하도록 한다.
- **어플리케이션 마스터**는 실행된 각 어플리케이션마다 존재한다. 최초 실행 후 스케줄러와 연결하여 실행을 위한 자원이 배정되기를 기다리고, 이후 실행중 자원 배정이 변경되면 현재의 실행 상황이 체크포인팅(checkpointing)되기를 기다린 이후 새로이 배정된 자원으로 작업을 다시 실행시킨다. 어플리케이션 마스터는 내부적으로 Horovod[13]를 이용해 여러 워커를 실행하여 분산 수행을 달성한다.

- **워커**는 Horovod에 의해 수행된 분산 프로세스로, TensorFlow 2.0 프레임워크에 기반하여 실제 딥 러닝 학습 작업을 수행한다. 워커 중 인덱스가 0인 워커가 치프 워커(Chief worker)인데, 요청에 따라 실제로 실행 상태를 체크포인팅하는 역할을 수행한다.

4.4. 스케줄러 구현

스케줄러는 어플리케이션 상태를 다음과 같이 구분한다.

- WAITING_FOR_INITIAL_CONTACT: 초기 상태이다. 어플리케이션 마스터가 실행된 이후, 어플리케이션 마스터가 스케줄러에 처음으로 연결할 때 까지의 상태이다. 어플리케이션 마스터가 스케줄러에 접속하면 WAITING_FOR_INITIAL_RESOURCE 상태로 전환된다.

- WAITING_FOR_INITIAL_RESOURCE: 스케줄러가 이 어플리케이션에 처음으로 자원을 할당할 때 까지의 상태이다. 자원이 할당되면 STANDBY 상태로 전환된다.

- STANDBY: 자원이 할당되었으나, 이 자원에 수행중인 다른 어플리케이션이 있어 대기중인 상태이다. 만약 배정받은 자원에 다른 어플리케이션이 없음이 확인되면 LAUNCHING 상태로 전환된다.

- LAUNCHING: GPU에 메모를 할당하고 TensorFlow 세션을 초기화하는 상태이다. 초기화가 완료되면 RUNNING_PROTECTED 상태로 전환된다.

- RUNNING_PROTECTED: 작업을 수행중인 상태이다. 이 상태에서 일정 시간 이상 머무르면 RUNNING 상태로 전환된다. 또

는, 이 상태에서 작업이 완료되면 FINISHED 상태로 전환된다.

- RUNNING: 작업을 수행중인 상태이다. 이 상태에서 작업이 완료되면 FINISHED 상태로 전환한다. 만약 자원 재배정이 발생하면 CHECKPOINTING 상태로 전환된다.
- FINISHED: 작업이 완료된 상태이다.
- CHECKPOINTING: 치프 워커가 체크포인팅을 수행중인 상태이다. 체크포인팅이 완료되면 STOPPING 상태로 전환된다.
- STOPPING: 현재 실행중인 Horovod 프로세스가 종료되기를 기다리는 상태이다. 이후 새로이 배정된 자원을 사용하기 위해 STANDBY 상태로 전환된다.

스케줄링 알고리즘이 작동될 때 STANDBY, LAUNCHING, RUNNING_PROTECTED, CHECKPOINTING, STOPPING 상태인 작업이 점유하고 있는 자원은 스케줄링이 불가능한 자원으로 취급되어, 이 자원을 다른 작업이 할당하거나, 이 자원을 사용하고 있는 자원에 새 자원을 할당할 수 없다.

자원이 재할당되면 작업은 일시적으로 종료되었다가 체크포인트를 기반으로 다시 실행되는데, 다시 시작되는 동안은 모델의 학습 과정이 진행되지 않는다. 자원의 재할당이 너무 자주 실행되어 작업 재시작으로 인한 오버헤드가 과도해지는 것을 막기 위해, 다음과 같은 규칙이 적용된다.

- 작업은 처음 실행되거나, 자원을 재배정받아 재시작한 후에는 RUNNING_PROTECTED 상태가 된다.
  RUNNING_PROTECTED 상태인 작업을 대상으로는 자원 재배정이 발생하지 않는다. RUNNING_PROTECTED 상태의 작업은 가장 최근의 LAUNCHING 상태에서 소모한 시간의 3배가 지난 후 RUNNING 상태로 진입한다.

- 스케줄링 알고리즘이 작동한 결과 제시된 새로운 자원 배정을 적용하였을 때, 실행중인 작업들의 처리율의 합이 초당 1 스텝 이상 향상되지 않는 경우 이를 적용하지 않는다. 다만, 새로운 자원 배정을 통해 기존에 실행하지 않았던 작업에 자원을 배정할 수 있는 경우에는 적용한다.

Figure 1은 스케줄러가 추적하는 어플리케이션 상태를 나타낸 상태도 (state diagram)이다.

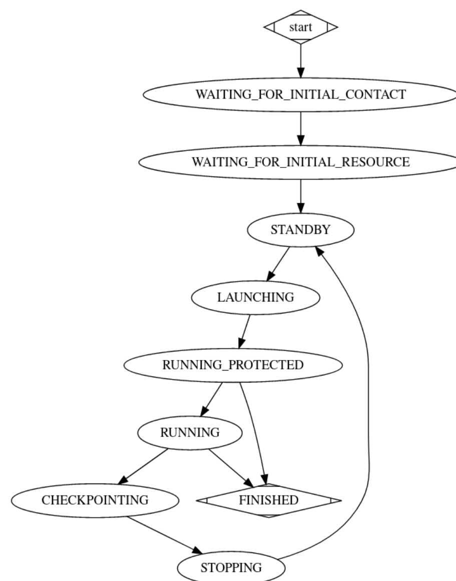


Figure 1: 어플리케이션의 상태

제 5절

성능 평가

이 단락에서는 앞서 설명된 실험적 구현체의 작동과 성능을 실증하기 위한 실험에 대해 설명한다.

5.1. 실험 설정

5.1.1. 실험 환경

실험은 한 대의 SuperMicro 4028GR-TRT 머신을 이용하여 진행되었다. 이 머신에는 정규 클럭이 2.10GHz인 Intel(R) Xeon(R) CPU E5-2695 v4 (2.10GHz) 프로세서 두 개, 총 251GiB RAM이 장착되어 있다. 6대의 NVIDIA TITAN Xp GPU가 장착되어 있으며, 각 GPU의 공유 메모리(shared memory) 총량은 12196 MiB이다.

5.1.2. 딥 러닝 모델

Convolutional Neural Network 기반의 모델 2가지를 이용하였다. 사용된 모델은 ResNet-50[14] 과 InceptionV3[15]이며, 155GB 크기의 ImageNet[16] 데이터를 이용하여 학습된다. 실험용으로 사용된 벤치마크 어플리케이션은 TensorFlow 벤치마크 [17] 에 있는 convolutional neural network를 위한 벤치마크 어플리케이션을 기반으로 개발되었다. 학습시 각 스텝에 GPU 전체를 통틀어 90개의 데이터 샘플이 사용되도록 설정했다. 여러 대의 GPU를 사용한다면, 이 90개 데이터 샘플을 각 GPU가 균등하게 나누어 학습하게 된다.

각 모델을 단독으로 GPU 개수를 변화시켜가며 단위 시간당 수행된 스텝의 수를 측정했으며, 이를 기반으로 스케줄러에 제공할 함수 f를 Figure 2와 같이 구성하였다.

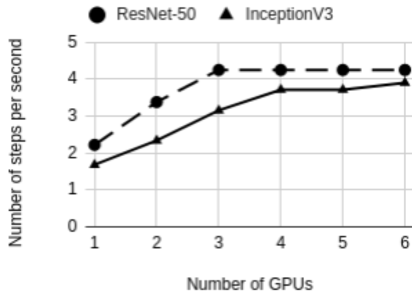


Figure 2: 모델에 대한 함수 f

생성 시점	모델	스텝 수
0초	ResNet-50	2,000
100초	InceptionV3	2,000
200초	ResNet-50	2,000
300초	InceptionV3	2,000

Table 1: 워크로드

5.1.3. 워크로드와 스케줄러

스케줄러의 작동과 성능을 확인하기 위한 워크로드는 Table 1 와 같다. ResNet-50, InceptionV3 두 모델 기반의 학습 어플리케이션이 각각 100초의 간격을 두고 생성되며, 이 전체 과정이 두 차례 발생한다. 이 때 각 어플리케이션은 2,000번의 스텝을 수행한 후 종료된다.

스케줄러의 성능을 비교하기 위해, 6개의 GPU를 크기가 n인 슬롯들로 정적 분할하여, 작업 하나를 슬롯 하나에 할당하는 스케줄러를 작성했다. v_{bound} 를 0.5 로 지정한 F-스케줄링 기반의 스케줄러와, n이 1, 2, 3, 6인 정적 분할 스케줄러에 대해 Table 1과 같은 워크로드를 실행했다.

5.2. 실험 결과

5.2.1. 소요 시간 분석

	F-Sched	n=1 Static	n=2 Static	n=3 Static	n=6 Static
소요시간(초)	1,350	1,557	1,593	1,498	2,138

Table 2: 워크로드 전체가 완료되기까지의 소요시간

전체 소요시간은 Table 2와 같이 F-스케줄링 기반의 스케줄러가 가장 빨리 작업을 끝냈고, 뒤이어 GPU 3개씩 하나의 슬롯으로 정적으로 분할한 스케줄러가 우수했다.

워크로드에서 수행된 작업별로 수행 시간의 조성을 분석하면 Figure 3과 같다. 정적 분할 스케줄러는 작업에 자원을 한번 할당하면 작업이 끝날 때까지 해당 자원을 재배정하지 않으므로, 작업을 GPU 자원에 실행할 때 소모되는 시간인 런칭 딜레이(Launching Delay)는 최소이다. 반면 F-스케줄링 스케줄러는 자원 재배정 후 추가적으로 발생하는 런칭 딜레이가 존재한다. 정적 스케줄러는 n의 값이 낮을수록 하나의 작업이 점유하는 자원의 양은 적지만 동시에 수행할 수 있는 작업의 개수는 늘어나는데, 따라서 n이 낮을수록 큐잉 딜레이(Queuing Delay)는 적어지나 작업의 실제 수행 시간이 상승한다.

이 분석 결과는 F-스케줄링이 주어진 워크로드에서 학습 수행 시간과 큐잉 딜레이를 종합적으로 줄이는데 우수했으나, 추가적인 런칭 딜레이라는 오버헤드가 발생했음을 뜻한다.

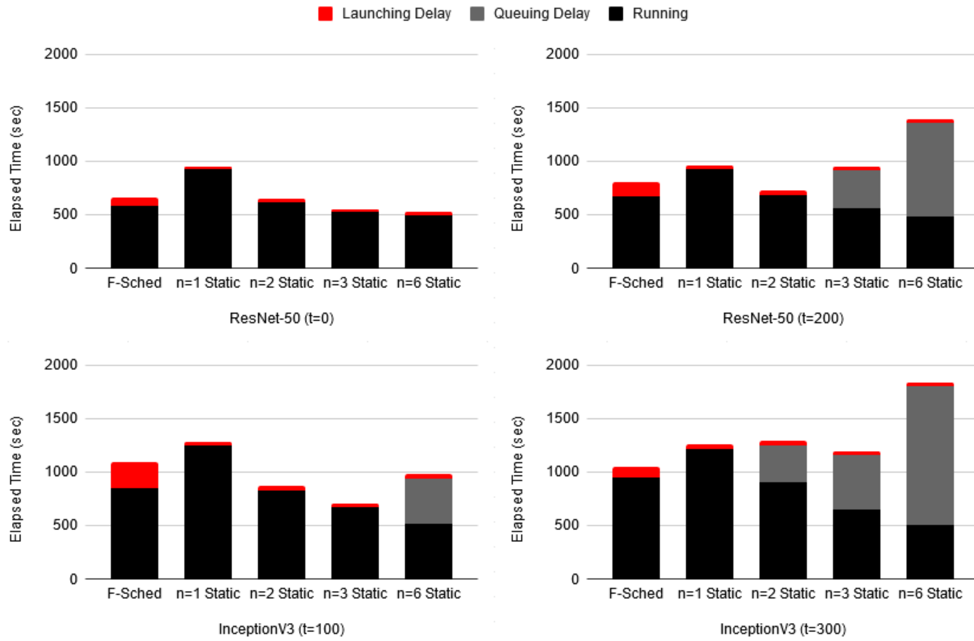


Figure 3: 워크로드에서 발생한 작업별 소요시간의 상세 분석. Queuing Delay는 작업이 WAITING_FOR_INITIAL_RESOURCE 상태에 있던 동안의 소요시간이다. Launching Delay는 작업이 STANDBY, LAUNCHING, CHECKPOINTING, STOPPING 상태에 있던 동안의 소요시간의 합이다. Running은 작업이 RUNNING 및 RUNNING_PROTECTED 상태에 있던 동안의 소요시간의 합이다.

5.2.2. GPU 자원 활용률

워크로드 수행 중 GPU 자원의 활용률을 누적분포함수로 표현하면 Figure 4와 같다. 이 때 자원의 활용률은 머신에 장착된 6개 GPU의 계산 자원 활용률의 평균치이다. 50%이상의 활용률 구간에 대해, F-스케줄링 스케줄러가 다른 모든 정적 분할 스케줄러에 비해 높은 GPU 활용률을 보여줌을 알 수 있다.

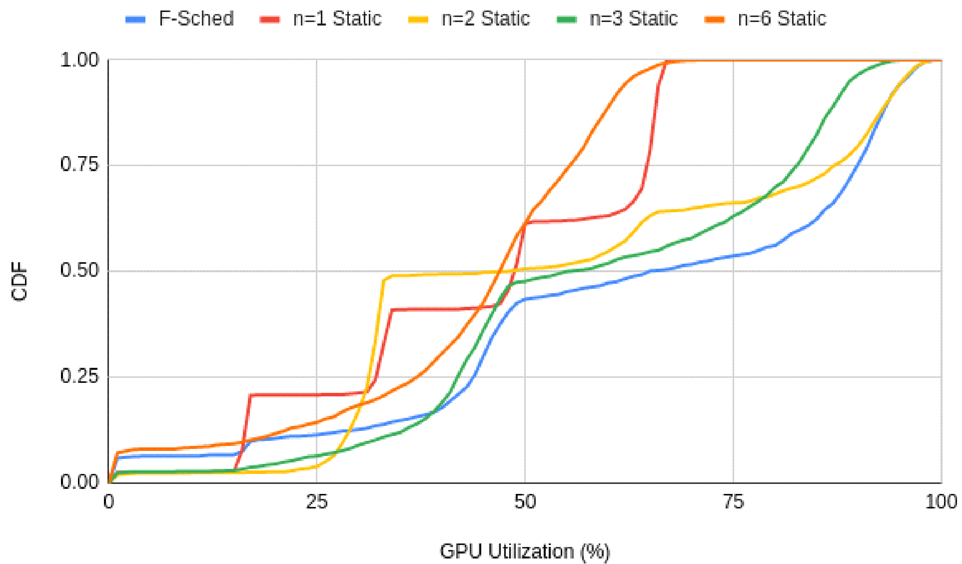


Figure 4: 워크로드 실행중 관측된 GPU 활용률의 누적분포함수

제 6절

결론

이 논문에서는 자원 관리자에 작업의 성능 특성을 나타내는 함수가 제시될 때, 자원 관리자가 보다 효율적인 스케줄링을 할 수 있는 가능성을 제시하고, 단순화한 자원 모델을 통해 시험적 구현체를 만들어 정적 분할 스케줄러와 성능을 비교했다.

성능 비교 결과, F-스케줄링 기반의 스케줄러는 학습 수행 시간과 큐잉 딜레이의 합을 줄이는 효과가 있었지만, 자원의 재배정마다 발생하는 추가적인 런칭 딜레이로 인한 손해 또한 발생했다. 이는 자원의 재배정이 실현되는 메커니즘을 개선하여 런칭 딜레이를 줄이는 방법에 대한 잠재적인 필요성을 제기한다.

한편, 한 번의 자원 배정이 오래도록 유지될수록 런칭 딜레이로 인한 손해 역시 더 큰 비율로 상환(amortize)된다. F-스케줄링은 새로운 작업이 추가되거나, 기존 작업이 완료되는 등의 이벤트에 반응하여 자원의 재배정 가능성을 검토한다. 따라서 스케줄링 시점에 추후 어떠한 이벤트가 발생할지 예측할 수 있다면, 앞으로 재배정을 덜 함으로서 발생하는 이득과, 당장 실행하기에 효율적인 자원을 배정함으로서 발생하는 이득 간의 적절한 균형을 찾을 수 있을 것이다.

다시 말하면, 개별 작업의 성능 특성을 이해하는 것이 스케줄링의 품질에 긍정적인 영향을 미치는 것처럼, 작업들이 모여 이루어진 워크로드로부터 스케줄링의 품질을 올릴 수 있는 특성을 규명할 필요 또한 발생할 수 있을 것이다.

참고문헌

[1] Amazon EC2 요금.
https://aws.amazon.com/ko/ec2/pricing/on-demand/.

[2] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In Proc. USENIX Annual Technical Conference (ATC), 2019.

[3] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu,N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao,Q. Zhang, F. Yang, and L. Zhou. Gandiva: Introspectivecluster scheduling for deep learning. In Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI), 2018.

[4] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal,M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth,B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In Proc. ACM Symp. on Cloud Computing (SoCC), 2013.

[5] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi,A. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In Proc. USENIX Symp. on Networked Systems Design andImplementation (NSDI), 2011.

[6] Hadoop: YARN Resource Configuration.
https://hadoop.apache.org/docs/r3.1.1/hadoop-yarn/
hadoop-yarn-site/ResourceModel.html

[7] Apache Hadoop 2.0.0-alpha Release Notes. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/release/2.0.0-alpha/RELEASENOTES.2.0.0-alpha.html.

[8] Placement Constraints. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/PlacementConstraints.html.

[9] A. Paszke, S. Gross, F. Massa, Adam Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proc. Advances in Neural Information Processing Systems (NIPS)*, 2019.

[10] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D.G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI)*. 2016.

[11] J. Gu, K. G. Chowdhury, M. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In Proc. USENIX Symp. on Networked Systems Design and Implementation (NSDI), 2019.

[12] V. J. Maccio, J. Hogg, and D. G. Down. On slowdown variance as a measure of fairness. Operations Research Perspectives, vol.

5, pp. 133-144, 2018.

[13] A. Sergeev and M. D. Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[14] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. *arXiv preprint arXiv:1512.03385* (2015).

[15] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the Inception Architecture for Computer Vision. *arXiv preprint arXiv:1512.00567* (2015).

[16] J. Deng, W. Dong, R. Socher, L. Li, K, Li, and L. Fei-Fei. ImageNet: A large-scale hierarchical image database. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2009.

[17] TensorFlow benchmarks. https://github.com/tensorflow/benchmarks.

Abstract

A Methodology for Efficient Scheduling on GPU-enabled Clusters

Jangho Seo

Department of Computer Science and Engineering

The Graduate School

Seoul National University

Deep learning training jobs utilize GPUs, which are relatively expensive resources on today's computing clusters. It is common to introduce a resource manager which governs multi-tenant GPU cluster shared among multiple jobs. This paper presents a protocol in which a resource manager can dynamically relocate a job to another set of resources, or elastically shrink or grow its resource usage. This paper also presents a case where the prototype implementation of the protocol outperforms a statically-partitioning scheduler in terms of GPU utilization and overall workload completion time.

Keywords: Resource manager, Scheduler, Resource elasticity, Deep learning, Distributed training, Resource utilization, Job completion time, Dynamic resource replacement

Student number: 2018-28102