Journal of The Institute of Electronics and Information Engineers Vol.59, NO.9, September 2022 https://doi.org/10.5573/ieie.2022.59.9.51
ISSN 2287-5026(Print) / ISSN 2288-159X(Online)

논문 2022-59-9-6

적응형 미니배치 데이터 재분배를 통한 멀티 GPU 환경에서 딥러닝 학습 응용의 효율성 향상

(Improving the Efficiency of Deep Learning Applications in a Multi-GPU Environment via Adaptive Mini-batch Data Redistribution)

김 인 모*, 김 명 선**

(Inmo Kim and Myungsun Kim©)

요 약

복수개의 GPU로 구성된 GPU 클러스터 내부에서 보통 다수의 DNN 모델 학습 응용이 실행된다. 이러한 환경에서 GPU 자원의 총량이 모든 학습 응용들이 필요로 하는 자원의 총량 보다 적다면 GPU 자원을 차지하기 위한 응용 사이의 경쟁은 피할 수 없다. 이때 발생한 경쟁의 정도와 각 학습 응용들이 각각의 GPU를 활용하는 정도에 따라서 어떤 응용들은 빠른 시간 내에 학습을 완료하고 어떤 응용은 이보다 훨씬 긴 시간 동안 학습을 진행해야한다. 본 연구에서는 이렇게 공정하지 못하게 학습 응용들에게 GPU들이 할당되는 것을 해결하는 알고리즘을 제안한다. 현재 각 응용들의 GPU 자원 활용 상태를 기반으로 예측된 학습 완료시간과 오프라인에서 각 응용들이 GPU를 독점해서 사용했을 때의 학습 완료 시간의 비율을 구해서 현재 그 응용의 현재 Slow-Down으로 정의한다. 현재 수행되는 학습 응용들의 Slow-Down 값이 비슷해지도록 각 학습 응용들의 GPU당 미니배치 데이터 비율을 주기적으로 조절하고 GPU 자원의 공정한 배분을 달성한다. 다양한 실험을 통하여 확인한 결과 최대 Slow-Down은 53%이상 감소하였고, 전체적인 GPU 활용율 역시 25% 증가하였다.

Abstract

More than one DNN model learning applications are usually executed inside a GPU cluster composed of several GPUs. In this environment, competition between applications to occupy GPU resources is inevitable if the total amount of GPU resources is less than that of resources required by all learning applications. Depending on the degree of competition that occurred at this time and the degree to which each learning application uses each GPU, some applications may complete learning in a short time and some must proceed for a much longer time than this. This study proposes an algorithm that solves the allocation of GPUs to learning applications in such an unfair way. Based on the current GPU resource utilization status of each application, the ratio of the predicted learning completion time and the learning completion time when each application uses GPU exclusively is calculated and defined as the current Slow-Down of the current application. The ratio of mini-batch data per GPU of each learning application is periodically adjusted and a fair allocation of GPU resources is achieved so that the Slow-Down values of currently executed learning applications become similar. As a result of confirming through various experiments, the maximum Slow-Down decreased by more than 53%, and the overall GPU utilization also increased by 25%.

Keywords : Fair-share scheduling, Deep learning applications, Multi-GPU, Slow-down

*학생회원, 한성대학교 IT융합공학과(Department of IT Convergence Engineering, Hansung University)
**정회원, 한성대학교 AI응용학과(Department of Applied Artificial Intelligence, Hansung University)
© Corresponding Author(E-mail : kmsjames@hansung.ac.kr)
※ 본 연구는 한성대학교 교내학술연구비 지원과제임.
Received ; February 22, 2022 Revised ; June 21, 2022 Accepted ; July 12, 2022

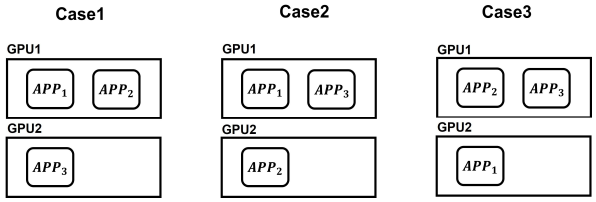


그림 1. 여러 DNN 학습 응용들이 두 개 이상의 GPU에 할당되는 경우 예시

Fig. 1. Example where multiple DNN applications are assigned to more than two GPUs.

Ⅰ. 서 론

최근 딥러닝의 기술적 발전은 DNN 모델을 사용한 응용의 활용 영역을 확장시키고 충분한 인식 정확성을 보장하고 있다. DNN 모델 자체의 연산 복잡도는 늘어가고 그 종류 또한 다양해져서 이러한 DNN 모델들을 학습할 때 DNN 연산에 매우 효과적으로 사용될 수 있는 SIMT(single instruction multiple thread) 기능을 제공하는 GPU가 많이 사용된다. 보통의 GPU 서버는 복수개의 GPU가 하나의 클러스터 형태로 제공되며 큰 모델을 학습할 때는 여러 개의 클러스터가 한꺼번에 동원되기도 한다.

클러스터 내부에서 다수의 DNN 모델 학습 응용을 실행할 때 사용할 수 있는 GPU 자원이 모든 응용들이 필요로 하는 자원의 총량보다 많다면 각각의 응용들은 필요한 만큼의 자원을 충분하게 할당받을 수 있다. 이는 응용들의 GPU 자원에 대한 경쟁을 발생시키지 않아서 GPU 자원에 대한 배분이나 할당에 관련된 이슈가 생기지 않는다. 하지만 클러스터 내부의 GPU 자원보다 응용들이 필요로 하는 자원의 총량이 더 많다면 응용들 간 GPU 자원에 대한 경쟁이 발생한다.

이때 어떤 응용들은 GPU 자원을 충분하게 사용할 수 있어 모델에 필요한 학습이 마치 GPU 하나를 독점해서 사용했을 때처럼 빨리 끝날 수 있고, 어떤 응용들은 GPU 자원에 대한 심한 경쟁으로 GPU 하나를 독점해서 사용했을 때보다 훨씬 더 긴 시간동안 학습이 진행될 수도 있다. 따라서 클러스터 내부의 복수개의 GPU를 다수의 DNN 모델 학습 응용에 효율적으로 할당해야 하는 이슈는 중요한 문제가 된다.

DNN 모델 학습 응용들의 현재 학습 진행 정도(x)를 GPU 하나를 독점해서 사용할 때의 학습 진행 정도(y)의 비율로 나타내면 어떤 응용들은 x/y의 값이 매우 크고 어떤 응용들은 상대적으로 매우 작을 수 있다. 각

표 1. 그림 1에 나타낸 응용들 각각의 완료시간

Table 1. Completion time for each of the applications shown in Figure 1.

	Solo-Run/ GPU	Case 1	Case 2	Case 3
APP_1	t	$2t$ / 2	$2t$ / 2	t / 1
APP_2	$2t$	$4t$ / 2	$2t$ / 1	$4t$ / 2
APP_3	$3t$	$3t$ / 1	$6t$ / 2	$6t$ / 2

응용들의 x/y 값을 비슷하게 유지할 수 있다면 이는 모든 응용들의 GPU 자원 사용정도를 공정하게 유지할 수 있는 방법이 될 수 있다.

본 논문에서는 딥러닝 모델을 학습시키는 모든 응용들의 x/y값을 비슷하게 유지시키는 것을 목표로 한다. 학습이 진행되는 동안 주기적으로 x/y 값이 작은 응용들이 사용하는 GPU 자원을 x/y 값이 큰 응용들에게 나눠줌으로써 각 응용들의 공정한 GPU 자원 활용을 보장해준다. 실험 결과 본 연구에서 제공하는 기법을 적용하기 전 대비 전체 응용의 x/y 최댓값과 최솟값의 차이는 53% 이상 감소하였고, x/y 값의 평균은 20% 이상 감소하였다.

Ⅱ. 연구 배경

1. 딥러닝 학습 응용들의 불공정한 GPU 자원 배분

그림 1은 3개의 딥러닝 모델을 학습하는 응용들이 2개의 GPU에서 실행될 때 각 GPU에 응용을 배치하는 경우를 예시한다. 표 1의 Solo-Run/GPU는 하나의 GPU에서 어떤 딥러닝 학습 응용 하나가 종료되는 시간을 나타내고 이때 응용 A는 시간 t가 걸렸음을 의미한다. 이때 응용별로 공정한 GPU 자원 분배가 이뤄졌는지 판단하기 위해 SD(Slow-Down)의 개념을 사용한다. 즉, Solo-Run/GPU 대비 실제 걸리는 시간의 비율이다. 표 1에서 예시한 Case 1에서의 APP_2는 $\frac{4t}{2t}=2$의 SD값을 갖고 이때의 SD값을 실제 걸리는 시간 $2t$ 옆에 나타내었다.

이상적인 경우, GPU 자원 배분이 완벽히 공정하게 이뤄진 상태라면, 모든 응용의 SD값은 같은 상태이다. 하지만 표 1에서 볼 수 있듯이 모든 경우에 있어서 어떤 응용은 2의 값을 갖고 어떤 응용은 1의 값을 갖게 되어 불균등함을 알 수 있다. 만일 세 응용 모두 비슷하게 1.66의 값을 갖는다면 보다 더 GPU 자원이 응용들에게 공정하게 분배된 상황이라 할 수 있다.

2. 딥러닝 학습 응용들의 GPU 자원 배분 방법

일반적으로 많이 사용되는 딥러닝 프레임워크인 PyTorch[1]에서 복수개의 GPU를 딥러닝 모델 학습에 사용하는 방법을 예를 들어 설명한다. PyTorch에서는 DataParallel이라는 API를 사용하여 딥러닝 모델 학습이 하나 이상의 GPU를 동시에 사용할 수 있게 한다. 입력 파라미터는 대상이 되는 딥러닝 모델과 수행하고자하는 GPU 인덱스이다.

DataParallel은 각 학습 과정의 각 이터레이션 마다 다음과 같은 과정을 따른다. 딥러닝 모델을 병렬처리에 사용할 각 GPU에 복사한다. 미니배치 데이터를 모델이 복사된 각 GPU에 동등한 비율로 나누어 할당한다. 각 GPU에서 Forward 과정 진행 후 출력 값이 나오면 각 GPU의 출력 값을 하나의 GPU 메모리에 모은다. 출력 값이 모이는 GPU를 Output Device라고 한다. 모인 값들을 이용하여 Output Device에서 손실함수의 기울기를 계산하고 병렬 처리하는 GPU에 기울기 값을 복사한다. 이후 각각의 GPU에서 기울기 값을 이용한 역전파가 진행된다. 마지막 과정으로 역전파의 결과로 나온 기울기 값을 Output Device에 모아서 모델을 업데이트 한다. 매개변수가 업데이트된 모델은 Output Device에만 존재한다. 따라서 다음 이터레이션에서는 Output Device로 지정된 GPU 메모리에 존재하는 모델을 각 병렬처리 대상의 GPU에게 복사해야 한다.

Ⅲ. 적응형 SD 기반 미니배치 재분배 기법

본 장에서는 딥러닝 학습 모델들이 공정한 방법으로 GPU 자원을 할당 받을 수 있는 SD 기반 딥러닝 학습 방법을 기술한다.

1. 시스템의 전체적인 동작방식

본 연구에서는 딥러닝 모델 학습 응용들의 GPU 자원의 공정한 사용 정도를 SD로 나타낸다. 또한 어떤 응용 APP_i의 SD값을 SD_i로 나타낸다. 동시에 실행되는 N개의 응용에 대하여 모든 응용의 SD값 집합을 $SD_{set} = \{SD_1, SD_2, \cdot\cdot\cdot, SD_N\}$이라고 하고, $1 \le i,j \le N$을 만족하는 임의의 원소 SD_i, SD_j에 대하여 그 차이가 가장 큰 것을 $SD_{\max}^{diff}$로 나타내면 식 (1)처럼 나타낼 수 있다.

$$SD_{\max}^{diff} = \max(\mid SD_i - SD_j \mid) \quad (1)$$

표 2. 4개의 GPU에서 두 개의 딥러닝 모델 학습 응용이 이루어지는 상황 예시
Table 2. Example where two DNN model applications are running on four GPUs.

	APP_1	APP_2
DR	[10, 0, 0, 0]	[0, 7, 3, 0]
Used GPU	GPU0	GPU1, GPU2
Mini Batch Data Ratio	GPU0 - 100%	GPU1-70% GPU2-30%

본 연구에서 제안하는 기법은 딥러닝 모델 학습 응용들이 공정하게 GPU자원을 사용할 수 있도록 $SD_i \in SD_{set}$를 만족하는 모든 SD_i를 일정한 간격으로 관찰한다. 이때 가장 큰 SD값을 갖는 APP_i는 가장 GPU 자원을 공정하지 못하게 할당받고 있는 상태로 볼 수 있다. 이를 해결하기 위하여 일정한 주기마다 시스템의 가용 GPU 자원의 현재 상태를 파악한 후 가장 큰 SD값을 가지는 응용에게 가장 최적으로 미니배치 데이터를 나누어 각각의 GPU에 할당하도록 한다. 이러한 방법을 통하여 가장 큰 SD값을 기록하는 응용의 SD값을 그때그때 줄여갈 수 있고 딥러닝 모델들의 학습 시간이 경과될수록 더 공정한 GPU 자원 배분의 결과를 기대할 수 있다.

2. 각 GPU에 할당되는 미니배치의 데이터 비율

매 이터레이션 마다 각 GPU에 할당된 미니배치 데이터의 비율을 DR(Data-Ratio)라고 정의한다. 이는 미니배치 데이터의 비율을 리스트 형태로 표현한 것이다. DR의 각 요소는 10 이하의 정수 값으로 구성되고, DR의 모든 요소의 합은 언제나 10이 된다. 예를 들면 [1, 2, 3, 4]는 GPU0~GPU3 순서대로 미니배치 데이터 비율을 1~4까지 갖고 있음을 나타낸다.

표 2는 4개의 GPU에서 두 개의 딥러닝 모델 학습 응용이 이루어지는 상황을 예시한다. App_1은 GPU0만을 사용하며 미니배치 데이터가 병렬처리 되지 않는 상태이다. App_2는 GPU1에 미니배치 데이터의 70%, GPU2에 30%, 그리고 GPU0과 GPU3은 사용하지 않음을 알 수 있다.

3. 딥러닝 모델 학습 응용의 SD 산출

SD_i는 응용의 DR_i에 따라 달라진다. 이를 정량화하면 식 (2)와 같고 SD_i는 매 에포크 종료 시점마다 값이 갱신된다.

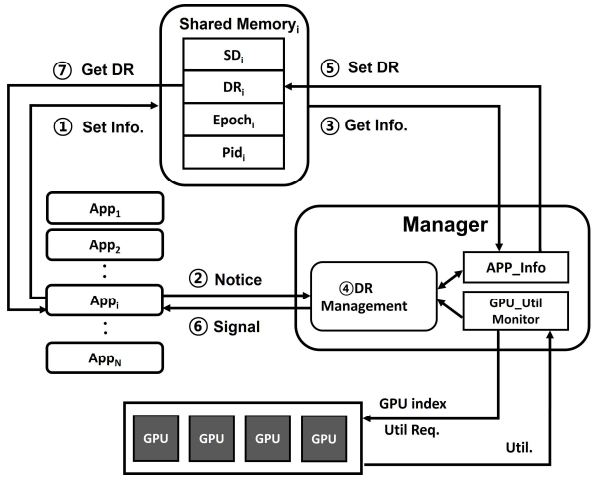


그림 2. 적응형 SD 기반 미니배치 재분배 전체동작
Fig. 2. Overview of adaptive SD-based mini-batch redistribution.

$$SD_i(DR_i) = \frac{T_{current} - T_{start} + iter\_left * iter\_time(DR_i)}{solorun\_time} \quad (2)$$

식 (2)에서 $solorun_time$은 단일 GPU에서 APP_i가 단독 수행 되었을 때 학습이 완료되기까지 걸린 시간을 의미한다. $T_{current}$는 현재의 시각, T_{start}는 응용의 시작 시점을 나타내므로 $T_{current} - T_{start}$는 측정시점까지 응용의 수행시간을 의미한다. $iter_time(DR_i)$는 DR_i 조건에서 현재 에포크에서 측정한 1이터레이션 수행 시 걸리는 평균시간이다. $iter_left$는 $T_{current}$부터 전체 학습이 완료되는 시간까지 남은 이터레이션의 수를 의미한다.

4. 적응형 SD 기반 미니배치 재분배

그림 2는 APP_i의 DR_i가 갱신되는 과정을 예시한다. 그림의 공유메모리를 통해 각 응용과 *Manager*가 데이터를 서로 주고받는다. 모든 응용은 고유의 공유메모리를 가지고 있고, n번의 이터레이션 마다 공유메모리에 현재 상태에 대한 SD_i값을 계산하여 기록한다. 본 연구에서는 5번의 이터레이션 마다 SD_i값을 기록하였다.

그림의 *Manager*는 기본적으로 $SD_{\max}$가 임계값을 넘는 경우, SD값이 가장 높은 APP_i의 DR_i을 재설정하여 SD_i를 낮추는 역할을 한다. DR_i 재설정 계산은 상황에 따라 GPU의 사용률 또는 응용의 SD값을 이용한다. 이때 각 GPU의 사용률은 그림을 통해서 알 수 있듯이 직접 GPU 디바이스 드라이버에게 요청하여 일정한 주기로 그 값들을 업데이트하고 기록한다. APP_i의 DR_i가 갱신되는 과정을 그림 2의 ①~⑦을 통하여 설명하면 다음과 같다.

APP_i는 현재 수행 중인 에포크가 완료되는 시점 즉, 에포크의 마지막 이터레이션에서 SD_i를 계산한 후, ① 공유메모리에 SD_i, DR_i, 현재 에포크 인덱스 그리고 프로세스 번호(*Pid*)를 기록한다. ② APP_i는 자신이 공유메모리에 기록을 끝냈음을 *Notice* 신호를 통하여 *Manager*에게 알린다. ③ *Manager*가 모든 응용의 공유메모리에 접근하여 기록된 값들을 가져와 APP_Info의 $APP_1 \sim APP_N$에 대한 정보를 갱신한다. 마지막 에포크를 끝낸 응용이 Notice 신호를 *Manager*에게 보내면 해당응용에 대한 정보가 APP_Info에서 삭제된다. ④ 그림 3의 알고리즘에 의해서 APP_i의 DR값 즉, DR_i를 산출한다. SD_i가 SD_{set}의 값들 중 최댓값이 아니거나, 현재의 $SD_{\max}$가 임계값보다 작은 경우에는 기존 DR을 유지한다(그림 3의 라인 11~13).

APP_i가 어떤 GPU상에서 겪을 수 있는 수행 정도의 느림과 빠름은 자신의 미니배치 데이터 비율뿐만 아니라 같이 수행되는 응용들의 미니배치 데이터 비율에도 영향을 받는다. 따라서 현재 APP_i 자신에게 할당된 GPU들 각각의 사용률과 할당되지 않은 GPU들의 사용률 역시 새로운 DR_i를 생성할 때 반드시 고려되어야 한다. 이를 세분화하면 다음의 두 가지 경우이다. 첫째로 i) 어떤 SD값이 큰 응용이 사용률이 높은 GPU에 할당되어 있고 그 GPU의 사용률과 가장 사용률이 낮은 GPU사이의 사용률 차이가 임계값 $Util_{th}$보다 클 경우, 두 번째는 ii) 그 차이가 $Util_{th}$보다 작을 경우이다. i)의 경우는 *DR_Update_Util*() 함수에 나타내었다. 먼저 이 함수는 사용률이 가장 낮은 GPU(그림 3의 라인 16 $GPU_{\min}$)를 찾는다. $GPU_{\min}$부터 현재 APP_i가 사용하는 GPU 중 가장 사용률이 높은 GPU까지 APP_i의 미니배치 데이터 비율이 감소하는 형태로 할당한다 (그림 4의 라인 2~11).

예를 들어 DR_i = [5, 5, 0, 0] 라면 App_i가 사용중인 GPU는 GPU0와 GPU1이다. GPU2가 $GPU_{\min}$이고 각 GPU0~2의 남은 활용률을 10%, 20%, 80% 이라고 가정한다. 이 때 GPU^i_{used}는 {GPU0, GPU1, GPU2}이다. 이때 각 GPU의 남은 활용률을 활용률의 총 합으로 나눈 값은 순서대로 0.09, 0.18, 0.72가 된다. 이 값들을 합이 10이 되는 DR_i로 변환 하면 [1, 2, 7, 0]이다.

Algorithm 1 DR Management

GPU^i_{used}: A set of GPUs utilized by App_i
GPU_{set} : $\{GPU_0, GPU_1, \dots GPU_m\}$, m GPUs in the system
n : Predefined period

1: **Function** DR_management (SD_i, DR_i, SD_{set})
2: **if** # of Apps ≤ m
3: GPU_{min} ← A GPU with the smallest Util. in GPU_{set}
4: **for** g in GPU_{set}
5: **if** g == GPU_{min}
6: new_DR_i[a] ← 10
7: **else**
8: new_DR_i[a] ← 0
9: GPU_{set} ← GPU_{set} - $\{GPU_{temp}\}$
10: **return** new_DR_i
11: **if** SD_i != max(SD_{set}) **or** $SD^{diff}_{max} < SD_{th}$
12: new_DR_i ← $current\_DR_i$
13: **return** new_DR_i
14: **else**
15: GPU_{max} ← A GPU with the biggest Util. in GPU^i_{used}
16: GPU_{min} ← A GPU with the smallest Util. in GPU_{set}
17: **if** (Util. of GPU_{max}- Util. of GPU_{min})> $Util_{th}$
18: new_DR_i ← DR_Update_Util(GPU^i_{used}, GPU_{set}, GPU_{min})
19: **else**
20: GPU_{dest} ← A GPU which has the minimum Avg. SD value in GPU_{set}
21: GPU_{src} ← A GPU indexed by the biggest value in DR_i
22: new_DR_i ←DR_Update_SD(SD_i, DR_i, SD_{set}, GPU_{src}, GPU_{dest})
23: **return** new_DR_i

그림 3. DR Management 알고리즘
Fig. 3. Pseudo-code of DR management.

ii)의 경우, APP_i가 속해 있는 GPU 중 가장 사용률이 높은 GPU(그림 3의 $GPU_{\max}$)와 전체 GPU중 사용률이 가장 낮은 GPU(그림 3의 $GPU_{\min}$)의 사용률 차이가 $Util_{th}$보다 작으면서 $SD^{diff}_{\max} > SD_{th}$ 상태를 나타낸다. 이러한 경우는 각 GPU 간 사용률 차이가 작기 때문에 SD값을 기준으로 새로운 DR_i를 찾아야 한다. APP_i가 사용하는 GPU 중 APP_i의 미니배치 데이터 비율이 가장 큰 것을 GPU_{src}로, GPU_{src}로부터 일정 미니배치 데이터 비율($0 \le r \le 10$)을 가져갈 GPU를 GPU_{dest}로 설정한다(라인 20~21). 이후 *DR_Update_SD()*함수는 r, GPU_{dest}, GPU_{src} 값들을 사용하여 새로운 DR_i를 구한다. 먼저 GPU_{src}의 미니배치 비율 당 APP_i의 성능 저하에 미치는 영향을 값으로 표현한 E_{mbr}를 계산한다. 이후 GPU_{dest}로 옮겨질 미니 데이터 비율 r을 SD_i와 SD_{set}의 최댓값과 최솟값의 평균값의 차이를 E_{mbr}로 나누어 구한다. 마지막으로 기존의 DR_i에서 GPU_{src} 인덱스 요솟값은 r만큼 빼고 GPU_{dest} 인덱스 요솟값은 r만큼 더하여 새로운 DR_i를 만든다(그림 4의 라인 14~18).

DR_i값이 변경되는 또 다른 조건이 한 가지 존재한다. App_i가 *Manager*에게 *Notice* 신호를 보냈을 때 학습이 진행 중인 응용의 개수가 시스템의 전체 GPU 개수와 같거나 적은 경우이다. 이 경우 1개 GPU에서 1개의 응용만 실행되도록 한다면 더 이상 모든 응용에게서

Algorithm 2 DR Update by Util or SD

GPU^i_{used}: A set of GPUs utilized by App_i
GPU_{set} : $\{GPU_0, GPU_1, \dots GPU_m\}$, m GPUs in the system
GPU_{min} : A GPU with the smallest Util. in GPU_{set}
GPU_{dest} : A GPU which has the minimum Avg. SD value in GPU_{set}
GPU_{src} : A GPU indexed by the biggest value in DR_i

1: **Function** DR_Update_Util(GPU^i_{used}, GPU_{set}, GPU_{min})
2: **if** $GPU_{min} \notin GPU^i_{used}$
3: GPU^i_{used} ← $GPU^i_{used} \cup \{GPU_{min}\}$
4: Util. remained ← 0
5: **for** j in GPU^i_{used}
6: Util. remained ← Util. remained + 100 - Util. of GPU_j
7: **for** k in GPU_{set}
8: **if** k in GPU^i_{used}
9: $new\_DR_i[k] \leftarrow \left\lfloor \frac{100 - Util.of\ GPU_k}{Util.remained} * 10 + 0.5 \right\rfloor$
10: **else**
11: new_DR_i[k] ← 0
12: **return** new_DR_i

13: **Function** DR_Update_SD(SD_i, DR_i, SD_{set}, GPU_{src}, GPU_{dest})
14: $E_{mbr} \leftarrow \frac{SD_i - 1}{DR_i[GPU_{src}]}$
15: $r \leftarrow \left\lfloor \frac{SD_i - (\max(SD_{set}) + \min(SD_{set}))/2}{E_{mbr}} \right\rfloor$
16: $new\_DR_i \leftarrow current\_DR_i$
17: $new\_DR_i[GPU_{src}] \leftarrow new\_DR_i[GPU_{src}] - r$
18: $new\_DR_i[GPU_{dest}] \leftarrow new\_DR_i[GPU_{dest}] + r$
19: **return** new_DR_i

그림 4. GPU 활용률 또는 SD값을 이용한 DR 업데이트 알고리즘
Fig. 4. Pseudo-code of DR update based on utilization or SD.

SD이 발생하지 않는다. 이때 App_i는 활용률이 가장 작은 GPU에서 미니배치 데이터를 100% 실행하도록 DR_i값이 변경된다. App_i 이후에 *Notice* 하는 다른 응용들은 App_i가 사용 중인 GPU를 제외하고 활용률이 낮은 GPU부터 순서대로 할당된다(그림 3의 라인 2~10). DR Management 알고리즘의 결과인 새로운 DR_i로 APP_Info의 APP_i 정보를 갱신한다. *Manager*는 ⑤ 공유메모리에 DR_i을 기록하고 ⑥ APP_i에게 *Signal*을 보내서 공유메모리의 DR_i이 갱신되었음을 알린다. ⑦ APP_i는 공유메모리에 접근하여 DR_i을 가져와 미니배치 데이터 비율을 수정하고 다음 에포크를 진행한다.

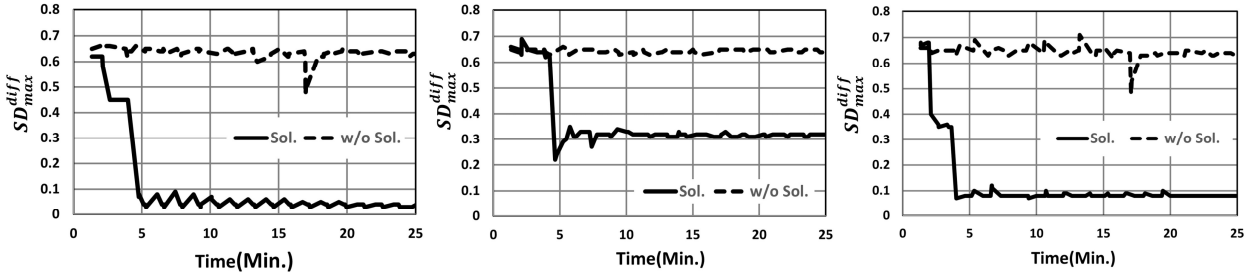
Ⅳ. 실 험

본 장에서는 DR Management 알고리즘을 적용하였을 때 모든 응용이 GPU 자원을 공정하게 할당 받는 것을 증명하는 실험 내용을 다룬다.

1. 실험환경

실험에 사용된 시스템은 표 3의 GPU 4개와 CPU 14개를 포함하는 리눅스 운영체제 기반 서버이다.

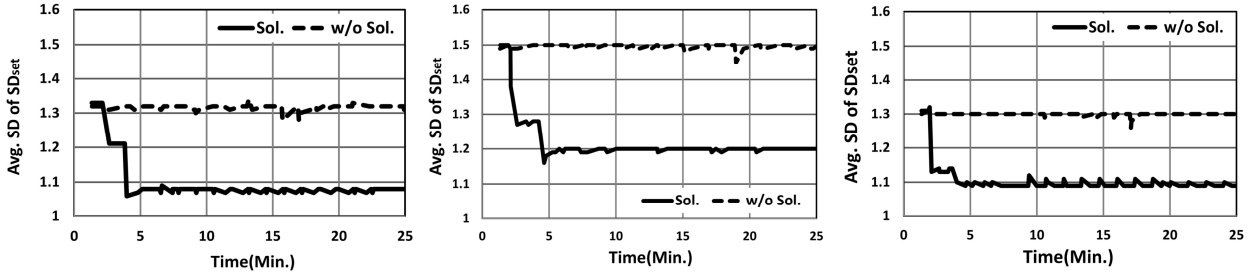
실험에서는 DenseNet121[2], ResNet50[3], AlexNet[4], ResNext[5], ShuffleNetv2[6], MnasNet[7] 총 6종류의 DNN 모델을 사용하여 결과를 확인한다.



(a) Dense x2, Res x2, Alex x2 (b) Dense x2, Res x2, Alex x1 (c) Dense x1, Res x1, Alex x1, ResNext x1, Shuffle x1, Mnas x1

그림 5. 학습 시간에 따른 $SD_{\max}^{diff}$ 변화

Fig. 5. Changes in $SD_{\max}^{diff}$ according to the learning time.



(a) Dense x2, Res x2, Alex x2 (b) Dense x2, Res x2, Alex x1 (c) Dense x1, Res x1, Alex x1, ResNext x1, Shuffle x1, Mnas x1

그림 6. 학습 시간에 따른 전체 응용의 SD 평균값의 변화

Fig. 6. Changes in average SD of all the applications according to the learning time.

이들 모델들은 실험부에서 표기의 간소화를 위하여 각각 Dense, Res, Alex, ResNext, Shuffle, Mnas로 나타낸다. 학습에 사용되는 데이터 셋은 ImageNet[8] 데이터 셋을 사용하였다.

표 3. 타겟 시스템 구성

Table 3. Target system configuration.

GPU (Quadro RTX 6000)	576 Tensor cores 24GB Memory 16.3 TFLOPS (FP32)
CPU (Intel i9-10940X)	14-core / 256GB Memory 19.25MB L3 Cache
OS	Ubuntu 18.04.4
CUDA ver.	CUDA 10.2

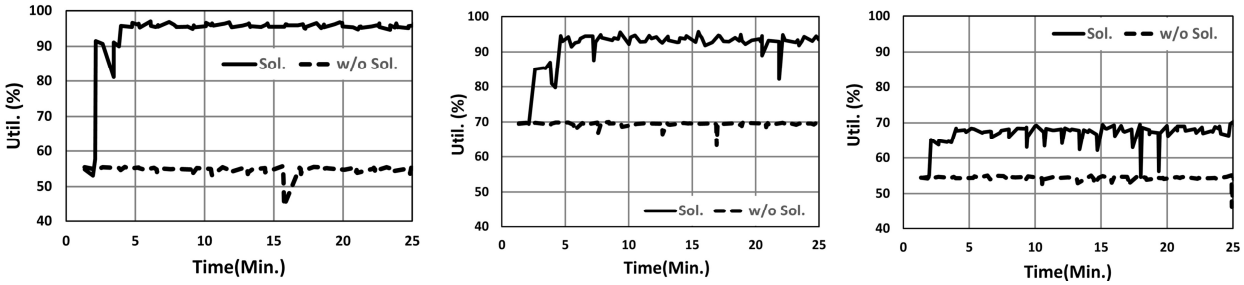
실험은 3종류의 DNN을 사용하여 4개 GPU에서 6개의 학습 응용(그림 5~7의 (a)), 3개 GPU에서 5개의 학습 응용(그림 5~7의 (b)), 그리고 6종류의 DNN을 사용하여 4개 GPU에서 6개의 학습 응용을 수행하였다(그림 5~7의 (c)). 3종류의 DNN만을 사용할 때는 Dense, Res, Alex를 사용하였다. 4개 GPU에서 6개의 응용 학습 시에는 각각 2개씩, 3개 GPU에서 5개의 응용 학습 시에는 Alex를 하나 제외하여 Dense 2개, Res 2개, Alex 1개를 사용하였다. GPU 활용률이 가장 적은 Alex하나를 제외하는 대신 사용하는 GPU의 개수를 하나 줄여서 전체 GPU 자원이 더욱 부족한 경우에도 공정한 자원의 배분이 이뤄지는지 비교하려고 한다. 실험의 결과를 표현한 그림 5~7에서 Sol.은 DR Management 알고리즘을 적용한 경우를 나타내고, w/o Sol.은 알고리즘을 적용하지 않은 경우를 나타낸다.

2. 실험 결과

실험결과는 다수의 응용이 동시에 실행된 후, 시간의 흐름에 따른 지표의 변화를 기준으로 분석한다. GPU 자원을 공정하게 배분 하였는지를 판단하기 위한 $SD_{\max}^{diff}$ 변화(그림 5), 전체적인 속도저하 개선여부를 확인하기 위한 평균 SD값의 변화(그림 6), 마지막으로 전체 GPU 활용률의 평균값의 변화를 살펴본다(그림 7).

가. 시간에 따른 $SD_{\max}^{diff}$ 변화

그림 5의 (a),(b),(c) 모두 알고리즘을 적용하지 않았



(a) Dense x2, Res x2, Alex x2 (b) Dense x2, Res x2, Alex x1 (c) Dense x1, Res x1, Alex x1, ResNext x1, Shuffle x1, Mnas x1

그림 7. 학습 시간에 따른 GPU 활용률 평균값의 변화
Fig. 7. Changes in the average value of GPU utilization according to the learning time.

을 때 학습 초기(0~5Min.) $SD_{\max}^{diff}$가 약 0.65의 값을 가지고 이 값이 학습이 진행되는 동안 비슷하게 유지된다. 제안된 알고리즘을 적용 시, 학습 초기의 $SD_{\max}^{diff}$는 미적용 시와 비슷하게 약 0.65의 값을 가진다. 하지만 학습이 진행되면서 실험 군 (a)는 $SD_{\max}^{diff}$가 0.1 이하까지 감소하고 (b)는 약 0.3, (c)는 약 0.1 까지 감소하여 학습 종료 시까지 감소된 값이 유지되는 것을 확인할 수 있다. 이것으로 (a)~(c) 세 개의 실험 군 모두 DR Management 알고리즘을 적용한 경우가 알고리즘을 적용하지 않은 경우 보다 더 공정하게 GPU 자원을 배분한다고 볼 수 있다. 실험 군 (a)와 (b)를 비교하면 GPU자원이 상대적으로 더 적은 (b)의 경우에도 알고리즘을 적용하면 최대한 모든 응용이 공정하게 GPU 자원을 배분받는 다는 것을 알 수 있다.

나. 시간에 따른 모든 응용의 평균 SD값의 변화

그림 6의 (a),(b),(c) 모든 경우에서 시작 시점에는 SD 평균값이 알고리즘을 적용했을 때와 적용하지 않았을 때의 값이 동일하고, 알고리즘 미적용의 경우는 시작 시점에서의 값이 거의 변화 없이 종료 시까지 유지된다. 하지만 제안된 알고리즘을 적용하면 실험 군 (a)는 약 1.1로 약 15% 감소, (b)는 약 1.2로 약 20% 감소, (c)는 약 1.1로 약 15% 감소하는 것을 확인할 수 있다. V-2-가 의 결과와 함께 분석해보면, 제안된 알고리즘을 적용 시 전체 DNN 학습 응용들의 학습이 진행되는 동안 SD값 차이가 줄어듦과 동시에 평균 SD값이 낮아짐을 알 수 있다. 이는 실행 중인 학습 응용들이 GPU를 공정하면서도 효율적으로 분배한다는 것을 의미한다.

다. 시간에 따른 전체 GPU의 평균 활용률 변화

마지막 실험으로, 제안된 알고리즘이 전체 GPU 활용률에 있어서도 더 효율적임을 밝히고자 한다. 그림 7에서 볼 수 있듯이 모든 실험 군에서 알고리즘 미적용 때와 비교하면 제안된 알고리즘을 적용 시 GPU 활용률의 평균값이 학습이 진행되는 동안 더 높음을 확인할 수 있다. 학습이 진행되는 동안 실험 군 (a)는 약 40%, (b)는 약 25% 높은 평균 GPU 활용률을 나타내었다.

(c)의 경우는 6개의 모델 중 ResNext, Shuffle, Mnas 모델이 포함되어 있다. 이들은 모델 자체의 연산 특성이 Dense, Res와 같은 모델에 비하여 GPU 내부의 SM(streaming multiprocessor)들을 단위 시간당 집중적으로 사용하지는 않는다[9]. 즉, 모델 자체적으로 GPU 활용률이 낮다. 따라서 (a), (b)의 경우보다 전체적인 GPU 활용률이 낮다. 하지만 이 경우에도 약 15% 더 향상된 평균 GPU 활용률을 나타내었다.

V. 결 론

본 논문에서는 다수의 DNN 모델 학습 응용들이 동시에 실행되는 환경에서 필요한 효율적인 복수 개의 GPU 활용 메커니즘을 제안하였다. 응용들이 DNN 모델 학습을 진행할 때 요구되는 GPU 자원이 클러스터 내부 GPU 자원보다 많은 경우 GPU 자원에 대한 경쟁이 필연적으로 발생하고 이 경쟁으로 인하여 GPU 자원의 공정한 분배가 이루어지지 못하는 문제가 발생한다. 본 연구에서는 이에 대한 해결책을 제시하였다. 어떤 응용이 실행되는 상태에서 현재 예측되는 학습 종료 시간과, 오프라인에서 이 응용이 한 개의 GPU를 독점으로 사용하였을 때의 학습 종료시간의 비율을 SD로 정의하고 이를 공정한 GPU 자원 배분의 지표로 사용

하였다. 응용들 사이의 SD값의 차이를 최대한 줄여서 GPU 자원을 최대한 공정하게 배분받도록 하였고, 각 학습 응용들의 각각의 GPU 상에서의 미니배치 데이터 비율을 주기적으로 조절하여 SD값 차이를 줄일 수 있었다. 이와 더불어 전체 응용들의 평균 SD값이 감소하면서 전체적인 GPU의 활용률이 증가되었다.

REFERENCES

[1] A. Paszke, S. Gross, F. Massa, A. 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. of Advances in Neural Information Processing Systems (NeurIPS), pp. 8024-8035, Vancouver, Canada, July 2019.

[2] G. Huang, Z. Liu., L. Van Der Maaten and K. Q. Weinberger, "Densely connected convolutional networks," in Proc. of IEEE Conf. on Computer Vision and Pattern Recognition(CVPR), pp. 2261-2269, Honolulu, July 2017.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition.," in Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR), pp. 770-778, Las Vegas, Nevada, June 2016.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in Proc. of Advances in Neural Information Processing Systems (NIPS), pp. 1106-1114, Lake Tahoe, December 2012.

[5] S. Xie, R. B. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in Proc. of IEEE Conf. on Computer Vision and Pattern Recognition(CVPR) pp. 5987-5995, Honolulu, July 2017.

[6] N. Ma, X. Zhang, H. T. Zheng, and J. Sun, "Shufflenet v2: Practical guidelines for efficient cnn architecture design," in Proc. of European Conf. on Computer Vision (ECCV), Vol. 14, pp. 122-138, Munich, Germany, September 2018.

[7] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard and Q. V. Le, "MnasNet: platform-aware neural architecture search for mobile," in Proc. of IEEE Conf. on Computer Vision and Pattern Recognition (CVPR), pp. 2820-2828, California, June 2019.

[8] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in Proc. of IEEE Conf. on Computer Vision and Pattern Recognition(CVPR), pp. 248-255, Miami, Florida, June 2009.

[9] J. Nickolls, and W. J. Dally, "The GPU computing era," IEEE micro, Vol. 30, No.2, pp. 56-69. April 2010.

저 자 소 개

김 인 모(학생회원)
2021년 한성대학교 IT응용시스템공학 학사 졸업.

<주관심분야 : GPU programing, DNN framework, Deep learning>

김 명 선(정회원)
2000년~2002년 LG전자 액세스 네트워크 연구소 주임연구원
2002년~2011년 삼성전자 DMC 연구소 책임연구원
2016년 서울대학교 전기컴퓨터 공학부 박사 졸업.
2016년~2019년 삼성전자 SR연구소 수석연구원
2019년~현재 한성대학교 IT융합공학부 조교수
<주관심분야 : 인공지능 시스템, Linux kernel, HW/SW Co-desgin, NPU 등>