석사학위논문

심층신경망을 위한 GPU 기반 시스템에서의 우선순위 기반 성능 격리 및 스케줄링 기법

2022년

한 성 대 학 교 대 학 원

I T 융 합 공 학 과

I T 융 합 전 공

조 호 진

석사학위논문
지도교수 김명선

심층신경망을 위한 GPU 기반 시스템에서의 우선순위 기반 성능 격리 및 스케줄링 기법

Priority-based Performance Isolation and Scheduling in GPU-based Systems for Deep Neural Networks

2022년 6월 일

한 성 대 학 교 대 학 원

I T 융 합 공 학 과

I T 융 합 공 학 전 공

조 호 진

석사학위논문
지도교수 김명선

심층신경망을 위한 GPU 기반 시스템에서의 우선순위 기반 성능 격리 및 스케줄링 기법

Priority-based Performance Isolation and Scheduling in GPU-based Systems for Deep Neural Networks

위 논문을 공학 석사학위 논문으로 제출함

2022년 6월 일

한 성 대 학 교 대 학 원

I T 융 합 공 학 과

I T 융 합 공 학 전 공

조 호 진

조호진의 공학 석사학위 논문을 인준함

2022년 6월 일

심사위원장 최 원석 (인)

심 사 위 원 서 화정 (인)

심 사 위 원 김 명선 (인)

국 문 초 록

심층신경망을 위한 GPU 기반 시스템에서의 우선순위 기반 성능 격리 및 스케줄링 기법

한 성 대 학 교 대 학 원
I T 융 합 공 학 과
I T 융 합 공 학 전 공
조 호 진

GPU 기반 시스템에서 CPU의 응용들은 DNN 연산을 GPU에 전달한다. 이때 CPU 측 OS에 의해 설정된 응용의 우선순위에 관계없이 DNN 연산은 FIFO 방식으로 GPU에 전달된다. 본 논문에서는 GPU 기반 시스템을 임베디드 시스템과 서버 시스템으로 나눠서 각 시스템에 맞는 GPU 스케줄링 기법을 제안한다. 제한된 환경을 갖고 있는 임베디드 시스템에선 CPU측의 높은 우선순위를 가진 DNN 응용이 GPU에서 최우선적으로 처리되어야한다. 본 논문에서는 임베디드 시스템에서 다수의 DNN 모델이 동시에 실행될 때 우선순위를 고려해서 GPU 리소스를 할당하는 스케줄링 기법을 제안한다. 낮은 우선순위의 DNN 응용이 먼저 실행되고 있어도 높은 우선순위의 DNN 응용이 이를 선점할 수 있어 높은 우선순위의 DNN 응용의 빠른 응답 특성을 보장한다. 다중 GPU로 구성되어있는 서버 시스템에서는 각 DNN 응용이 우선순위에 비례한 GPU 점유율을 가질 수 있는 gCFS를 제안한다. gCFS는 CPU 측의 공정 공유 스케줄링(fair-sharing scheduling) 을 계승하여 우선순

위에 비례하여 성능 분리를 달성한다. 스케줄링 단위를 세분화하여 GPU 점유시간을 보다 정확하게 제어할 수 있으며 DNN 워크로드를 더 조밀하게 대기열에 넣을 수 있어 GPU 유휴 시간을 단축할 수 있다. 스케줄링 시 DNN 워크로드의 길이는 주어진 GPU 점유시간으로 조정되고 최적의 GPU 선택이 동적으로 수행된다. 각 시스템에서 다수의 DNN 모델이 동시에 실행될 때 스케줄링 기법이 존재하는 경우가 존재하지 않는 경우에 비해 성능 향상이 크게 개선되었으며 전체 실행 시간이 각각 최대 76.6%, 40.4% 단축된다.

【주요어】 다중 DNN, 우선순위, GPU 공유, 성능 격리

목 차

표 목 차

그 림 목 차

제 1 장 서론

기계번역, 사물탐지, 의료진단, 자율주행 등 인공지능 서비스의 확산은 딥러닝 기술의 발전에서 비롯됐다. 일반적으로 딥러닝은 계산 집약적이며 훈련 시 동일한 DNN(Deep Neural Network) 모델에 대하여 서로 다른 데이터 셋을 동시에 처리해야하고 추론 시에도 서로 다른 DNN 모델을 처리해야한다. 이에 따라 DNN 연산에 적합한 하드웨어 가속 장치가 필요하며 GPU는 그러한 목적에 알맞은 컴퓨팅 장치로 간주된다.

모바일에서 거대한 데이터 센터에 이르는 GPU 클러스터를 포함하여 리눅스는 고품질의 다중 작업 기능 및 다양한 패키지를 제공하기 때문에 핵심 운영체제로 사용되어 왔다. 리눅스의 CFS(completely fair scheduler)는 우선순위 기반 성능 격리를 제공하며 [1], 각 응용의 CPU 점유시간은 주어진 우선순위에 비례한다. 사용자 레벨의 응용에서 각 응용의 우선순위는 −20에서 19 사이의 nice 값(nice value)을 통해 결정된다. 이 값은 리눅스 내에서 미리 정의된 가중치로 매핑된다. 여기서 −20은 가장 높은 우선순위(가중 큰 가중치)를 나타내고 19는 가장 낮은 우선순위(가장 작은 가중치)에 매핑된다.

그러나 이러한 nice 값 기반의 성능 격리는 CPU 측면에서만 작동하며 GPU 측면에서는 DNN 응용의 nice 값에 관계없이 FIFO(first in first out)로 동작한다. DNN 연산을 담당하는 GPU가 CPU 측의 성능 격리를 위한 동작들을 감지하지 못한다는 의미이다. 또한 CPU와 달리 GPU는 비선점 장치이며 모놀리식(monolithic) 리소스이기 때문에 DNN 응용에서 GPU에 대한 세분화된 제어 기능을 가질 수 없다. 이러한 GPU 기반 시스템에는 크게 임베디드 시스템과 서버 시스템 2가지로 분류 되며 각각의 사용 목적과 특징이 다르다. 이에 따라 본 논문에서는 각 시스템에 알맞은 스케줄러를 제안한다.

임베디드 시스템은 자율주행, 드론, 로봇 등에 활용되며 이는 임베디드 시스템이 크기가 작아 휴대성이 좋으며 낮은 소비전력을 가진다는 특징이 있기 때문이다. 하지만 고성능의 하드웨어로 이루어진 서버에 비해 낮은 성능과 메모리에 한계가 있다. 이에 따라 제한된 임베디드 시스템 환경에서 여러 종류

의 DNN 모델들을 적절한 시간 안에 실행을 완료하는 것이 중요하게 여겨진다. DNN 모델들은 서로 다른 타이밍에 연산을 요구할 수 있고, 동시에 연산을 처리해야 할 수도 있다. 이때, 우선순위가 높은 응용이 사용하는 DNN 모델들은 우선순위가 낮은 응용이 사용하는 DNN 모델보다 DNN 연산을 위한 GPU 자원을 우선적으로 할당받을 수 있어야한다.

참고문헌 [2]의 선행 연구에서는 DNN 응용을 2개의 우선순위 클래스로 분류하여 높은 우선순위 클래스에 속한 DNN 응용은 세부적으로 서로 다른 우선순위가 부여되고 낮은 우선순위 클래스에 속한 DNN 응용은 모두 동일한 우선순위를 갖게 된다. 부여된 우선순위에 따라 차등적으로 GPU 자원을 점유할 수 있다.

선행 연구에서의 우선순위는 CPU 측면에서의 우선순위를 의미하고 GPU에 전달된 커널(kernel) 수준에서는 우선순위가 고려되지 않는다. 본 연구에서 임베디드 시스템용 우선순위 기반 다중 DNN 스케줄러는 CFS의 nice 값을 이용하여 보다 세분화된 우선순위를 제공하며, GPU 내부에서는 전달된 커널에도 2개의 우선순위를 부여할 수 있다. 이는 우선순위가 높은 DNN 응용에 GPU 리소스를 우선적으로 할당하는 목적에 맞는 우선순위 기반 스케줄링이 가능하다.

임베디드 시스템 환경에서 이러한 DNN 모델들 간의 스케줄링 특성이 가능하려면 두 가지 사항을 충족해야 한다. 첫째로 사용자가 DNN 모델에게 부여한 nice 값 즉, 우선순위에 기반을 둔 DNN 연산용 GPU 자원 스케줄링이 가능해야 한다. 둘째로 리소스 선점이 불가능한 CFS와 달리, 먼저 수행중인 낮은 우선순위의 DNN 모델이 나중에 도착한 높은 우선순위의 DNN 모델에 선점(preemption)되어야 한다. 본 논문에서는 이러한 두 가지 사항을 만족하는 임베디드 GPU 시스템용 우선순위에 기반을 둔 다중 DNN 모델들의 스케줄링 기법을 제안한다.

서버 시스템은 높은 추론 정확성을 위해 DNN 연산 규모는 점점 커지는 추세와 더불어 여러 사용자들의 DNN 연산 요구량을 충족하기위해 여러 GPU로 구성되어있는 클러스터 형식으로 이루어져있다. 그렇기 때문에 GPU 기반 서버 시스템의 성능을 극대화하기 위해서는 효율적인 GPU 공유

(GPU-sharing)가 관건이다. 여러 DNN 응용에 걸쳐 GPU를 정적으로 분할하면 이상적인 성능 격리를 제공할 수 있지만 실제로는 GPU 활용률을 저하시킨다. DNN 응용을 특정 GPU에만 할당하는 경우, 헤드 오브 라인 블로킹(head-of-line blocking) [3]이 불가피하여 GPU의 효율적인 활용을 방해한다. 참고문헌 [4], [5], [6]과 같은 많은 선행 연구는 DNN 모델 완료 시간에 최적화된 GPU 스케줄러를 제공한다. 또한, 참고문헌 [7], [8]은 클러스터에서 GPU의 공정한 공유를 위한 수단을 제시한다. 공정 공유 GPU 스케줄링 관점에서 대부분의 이전 연구는 미니 배치 크기의 작업에 GPU 리소스를 동일하게 사용할 수 있도록 하는 효과적인 스케줄링 기술을 제공한다. DNN 응용의 진행 단계를 모니터링하면서, 그 시점에서 뒤쳐진 응용에게 더 많은 GPU 사이클이 주어진다.

공정 공유 스케줄링(fair-share scheduling) 정책은 함께 실행되는 응용의 가중치를 기반으로 시스템 리소스를 제공한다. 이 스케줄링 정책은 SLO(service-level objective)에 지정된 필수 컴퓨팅 리소스를 안전하게 보호한다. 이러한 이유로 공정 공유 스케줄링은 성능 격리를 제공할 수 있는 효과적인 방법이다 [9, 10]. 각 DNN 응용에 서로 다른 우선순위(가중치)가 주어지고 이에 비례하는 GPU 점유율을 가질 수 있다면 GPU 클러스터 내에서 진정한 성능 격리를 제공할 수 있다.

본 논문에서는 gCFS라는 다중 GPU를 사용하는 서버 시스템을 위한 DNN 스케줄러를 제안한다. gCFS는 각 DNN 모델에 주어진 nice 값에 따라 DNN 모델이 스케줄링 라운드에서 GPU를 점유할 수 있는 타임 슬라이스를 결정한다. 작업을 스케줄링 할 때, gCFS는 클러스터 내에서 가장 적절한 GPU 리소스를 선택하고 nice 값에 의한 타임 슬라이스가 계산되면 프로파일링을 통한 추정된 레이어의 실행 시간을 이용하여 이 스케줄링 라운드에 GPU를 사용할 수 있는 DNN 모델의 레이어 수를 결정한다.

더욱이 gCFS의 스케줄링 단위는 하나의 DNN 모델이나 미니 배치가 아니라 여러 개의 연속된 레이어로 구성된 작업이다. 이러한 방식으로 DNN 워크로드는 더 짧은 간격으로 GPU로 전송된다. 따라서 여러 DNN 모델이 동시에 실행되면 각 DNN 모델의 완료 시간이 단축된다.

즉, gCFS는 이전 연구와 달리 GPU의 우수한 병렬 처리 능력을 유지하면서도 DNN 응용이 CPU 측 CFS와 마찬가지로 GPU를 사용할 수 있는 새로운 성능 격리 기술을 제공한다. 또한 사용자의 개입 없이 서로 다른 타이밍에 도착하는 DNN 워크로드를 클러스터 내부의 GPU에 동적으로 할당할 수 있으며, 헤드 오브 라인 차단을 제거하기 위해 스케줄링 작업은 모델 전체 또는 미니 배치 기반이 아닌 레이어 집합 단위로 수행되므로 GPU 클러스터는 항상 처리해야 할 작업이 존재하여 유휴(idle) 상태로 있는 시간이 단축된다.

본 논문의 2장에서는 연구 배경과 앞서 간략히 설명하였던 GPU의 특징들로 인하여 GPU 기반 시스템에서 발생하는 문제점에 관해 설명하고 3장과 4장에서는 2가지의 시스템에 대하여 제안된 스케줄링 기법을 설명하고 실험과 분석을 진행한다. 마지막으로 5장에서 결론을 정리한다.

제 2 장 연구 배경 및 문제점

본 장에서는 CPU 측의 우선순위에 의해 결정된 가중치에 따라 CPU 사용 시간을 제공하기 위해 기존의 리눅스에서 구현된 CFS와 GPU 기반 시스템에 관해 설명한다. 그럼 다음 해결해야 할 문제를 정의한다.

제 1 절 Linux Completely Fair Scheduler

리눅스 2.6.23 릴리스 이후, 리눅스는 completely fair scheduler(CFS) [11]를 채택하였으며 CFS는 만족스러운 QoS 결과를 제공하는 공정 공유 스케줄러로 활용되었다. QoS 요구사항을 충족하기위해 CFS는 각 작업의 QoS 수준을 나타내는 사전 정의된 작업의 가중치에 따라 CPU를 점유할 수 있게 함으로써 작업 간의 공정한 분배를 가능하게 한다. CFS는 가중 라운드 로빈(weighted round-robin) 방식으로 작업을 스케줄링하기 때문에 타임 슬라이스(time slice)의 개념을 사용한다 [12]. 작업은 선점 없이 일정한 시간동안 CPU를 사용할 수 있으며, 우리는 그것을 타임 슬라이스라 한다. 타임 슬라이스는 지정된 시간 동안 작업의 물리적 실행 시간을 의미하므로 작업의 모든 타임 슬라이스를 더하여 작업의 CPU 점유시간을 계산할 수 있다. CFS에서 어떤 한 작업 τ_i의 타임 슬라이스는 다음 식을 통해 얻을 수 있다.

$$TS_i = \frac{w_i}{\sum_{\tau_i \in Q_k} w_j} \times C \quad (1)$$

여기서 w_i는 τ_i의 가중치이고 가중치는 τ_i에게 부여되는 −20과 19 사이의 nice 값(nice value)으로 결정되며 낮은 nice 값이 높은 가중치에 대응된다. Q_k는 k번째 CPU에서 스케줄링되는 τ_i를 포함한 작업들의 집합을 나타내며, C는 미리 정의된 스케줄링 라운드 간격이다 [1].

제 2 절 CFS의 virtual runtime과 부하 재분배(load balancing)

CFS는 각 CPU 당 하나의 런큐를 제공하며 각 런큐는 RB 트리 [1]로 구성되어 있으며 트리의 노드는 작업을 의미한다. 트리에서 작업의 위치는 작업의 virtual runtime에 의해 결정되며, τ_i의 virtual runtime은 다음 식을 통하여 얻어진다 [1].

$$vr_i(t) = \frac{w_0}{w_j} \times C_i(t) \quad (2)$$

여기서 w_0은 CFS의 nice 값 0에 해당하는 가중치를 뜻하며, $C_i(t)$는 시간 t일 때 τ_i의 실제 누적 실행 시간을 의미한다. 위의 식으로부터 알 수 있듯이, τ_i의 가중치가 클수록 τ_i의 virtual runtime은 작아져 런큐의 가장 왼쪽 노드에 위치 할 가능성이 높아지고, 가장 왼쪽 노드의 τ_i는 현재 CPU에서 실행되고 있는 작업이 CPU를 포기한 직후에 CPU로 스케줄링 된다. 즉, $w_i \leq w_j$ 일 때 $C_i(t) \leq C_j(t)$이 성립한다. 게다가 E_i 를 τ_i 가 완료될 때까지의 총 실행시간이라고 정의했을 때 CPU 점유시간이 길수록 작업의 총 실행시간이 짧아지기 때문에 $E_j \leq E_i$ 도 만족한다.

CFS는 다중 CPU에서 부하(load)를 균등하게 분산시키기 위해 부하 재분배(load balancing)를 주기적으로 수행하며, CPU의 부하를 다음과 같이 정의한다 [1].

$$Load_k = \sum_{\tau_j \in Q_k} w_j \quad (3)$$

여기서 Q_k는 k번째 CPU의 런큐를 뜻한다. CPU간의 로드 불균형이 임계값 이상인 경우 CFS는 부하가 가장 큰 런큐에서 가장 작은 런큐로 일부 작업을 이주(migration)시켜 런큐 간의 부하의 균형을 맞춘다. 부하 재분배를 통해

CFS는 CPU당 가중치 기반 공정 분배를 시스템 전체 CPU에 대한 공정 분배로 확장 가능하다.

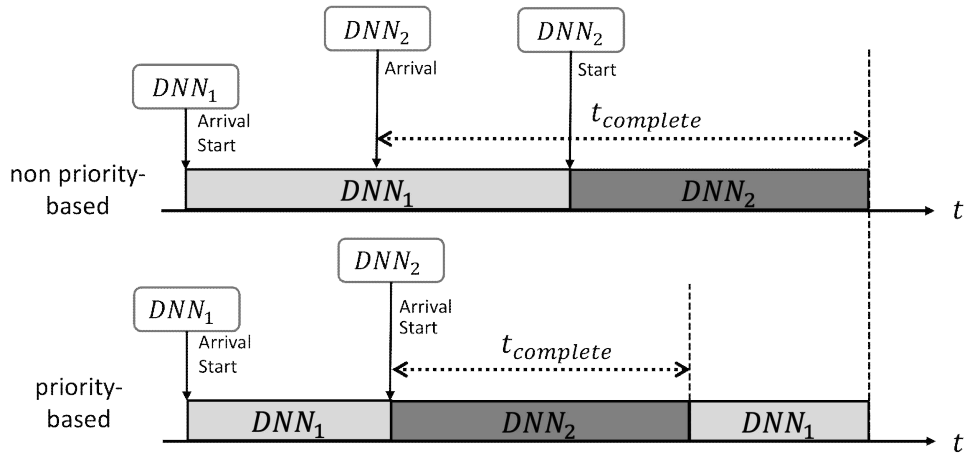
제 3 절 다중 DNN 연산을 위한 GPU 기반 시스템에서의 문제점

클라이언트 장치인 GPU는 호스트 CPU로부터 DNN 연산을 수행하는 커널(kernel)을 전달받는다. DNN의 레이어는 하나 이상의 GPU 커널로 구성되며, 하나의 커널은 사용자에 의해 지정된 그리드 크기로 GPU의 EE (execution engine) 큐에 들어간다 [13]. 그 후 그리드 내의 모든 스레드 블록이 SM(streaming multiprocessor)에서 연산을 수행한다. 각 커널은 FIFO 순서에 따라 EE 큐에서 대기하며 커널을 런치(launch)한 호스트 측의 작업의 우선순위에 관계없이 EE 큐에 도착한 순서대로 SM을 점유한다.

호스트에서 DNN 작업의 CPU 점유시간은 CFS가 제공하는 가중치에 따라 식 (2)에 나타난 virtual runtime에 의해 제어된다. 따라서, 우선순위가 높은 작업은 우선순위가 낮은 작업보다 CPU를 점유 할 수 있는 기회가 더 많이 주어지고 GPU에 커널 실행을 더 요청 할 수 있다. 그러나 우선순위가 낮은 커널이 GPU EE 큐에 먼저 도착했다면 우선순위가 높은 커널은 이전 커널의 실행이 완료될 때까지 기다려야하며 이에 따라 우선순위가 높은 커널의 실행 시간 지연이 발생한다. 정리하면, CPU 관점의 DNN 작업의 가중치가 GPU 쪽에 전달되지 않는다.

1) 임베디드 시스템의 특징과 문제점 정의

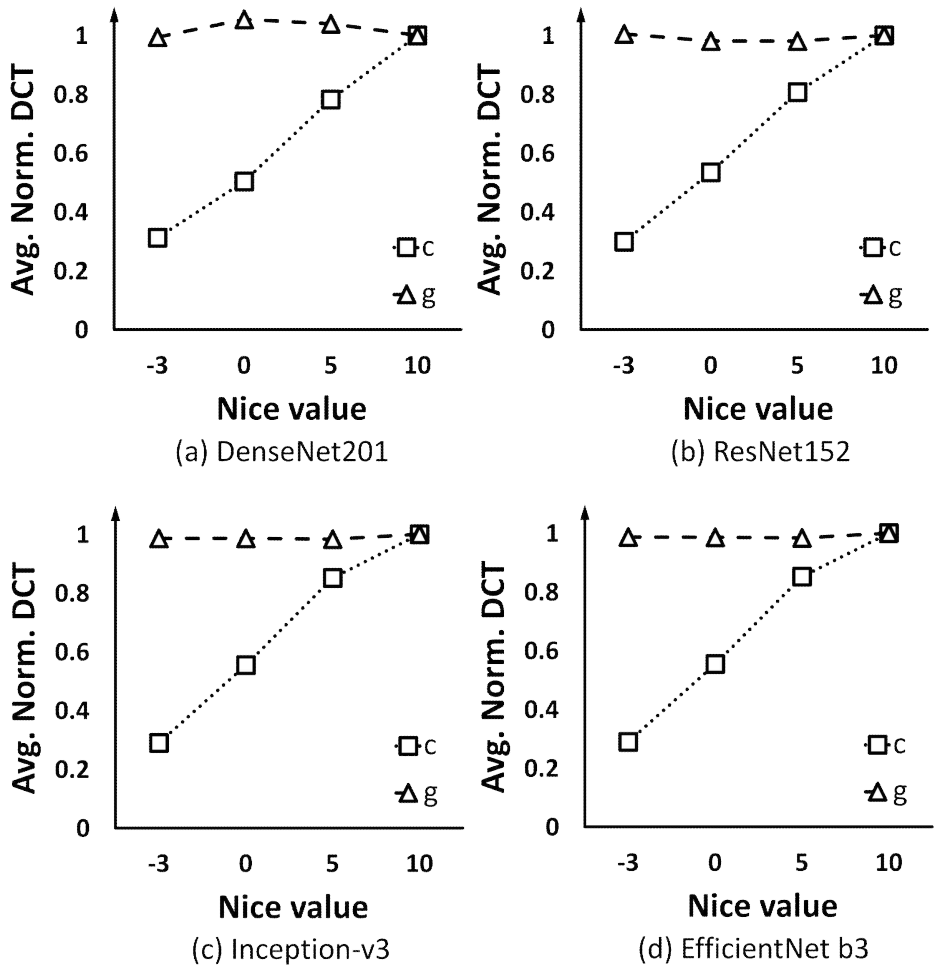
임베디드 시스템은 시스템 설계할 때부터 특정 기능을 수행할 목적으로 만들어지는 시스템으로 기능에 필요한 최소한의 하드웨어로 구성된다. 그렇기 때문에 다수의 GPU를 클러스터 형식으로 사용하는 서버 시스템과 달리 대부분 단일 GPU를 사용하며 메모리나 전력 등 제한적 환경을 가지고 있다. 또한 임베디드 시스템은 특정 기능을 적절한 시간 안에 수행을 완료해야하는 특성이 요구된다. 즉, 제한된 환경 내에서 앞서 설명한 GPU 기반 시스템의 문제점을 해결하려면 우선순위가 높은 DNN 작업이 우선순위가 낮은 DNN



[그림 2-1] 임베디드 시스템에서 문제점

작업보다 더 빨리 완료되어야한다. 이를 위해서는, CFS의 가중치가 큰 즉, 우선순위가 높은 DNN 작업에게 GPU 자원을 우선적으로 할당하는 것이 중요하다.

[그림 2-1]은 DNN 연산용 GPU 자원을 우선순위를 고려하여 DNN 모델들에 할당하는 경우와 그렇지 않은 경우의 차이를 나타낸다. [그림 2-1]에서 DNN_1 보다 DNN_2 가 더 높은 우선순위의 DNN 모델이며 DNN_1 이 도착하고 실행되는 도중에 DNN_2 가 도착한 경우를 나타낸다. 이 때 $t_{complete}$ 는 DNN 모델이 도착했을 때부터 수행이 끝날 때까지 걸린 시간을 의미한다. [그림 2-1]의 윗부분에서 알 수 있듯이, CFS의 가중치가 GPU 로 전달되지 않으면 높은 가중치 즉, 높은 우선순위의 DNN_2 가 순차적으로 DNN_1 의 연산이 완료된 후 실행이 시작된다. 반면, 그림의 아래 부분은 우선순위를 고려한 경우의 그림이다. 낮은 우선순위를 가진 DNN_1 이 먼저 도착해서 실행중이지만 우선순위가 높은 DNN_2 가 도착하면 GPU 자원을 DNN_2 에게 먼저 할당하여 DNN_2 의 실행이 완료하고 나서 남아 있는 DNN_1 의 연산을 실행한다. 따라서 $t_{complete}$ 가 짧아져 우선순위가 높은 DNN 모델들에게 GPU 자원을 우선적으로 할당하여 응답특성이 향상될 수 있다.



[그림 2-2] 서버 시스템에서 문제점

2) 서버 시스템 특징과 문제점 정의

임베디드 시스템과 달리, 서버 시스템은 모든 면에서 개별적인 하드웨어 성능이 우수하며 보통 다중 GPU로 구성되어 있어 여유로운 리소스를 활용하여 빠른 DNN 연산이 가능하다.

서버 시스템에서 앞에서 언급한 문제점을 직관적으로 설명하기 위해 하나의 실험을 [그림 2-2]에서 보여준다. 그림은 GPU 1개, CPU 1개만 사용하여 4개의 DNN 모델이 동시에 실행되며 각 모델은 100회 이상 실행된다. 그림은 nice 값이 서로 다른 동일한 DNN 모델의 평균 완료 시간을 나타내며 그

림의 모든 평균 완료 시간 값은 nice 값 10의 평균 완료 시간 값으로 정규화 되고, 각 그림의 nice 값 10의 결과는 1이다. 그림에서 범례 c는 오직 CPU만을 사용하여 DNN 모델이 실행되고 g는 GPU에 의해 실행되는 경우이며, 그림에서 보여주듯이 4개의 DNN 모델이 모두 CPU로 연산 할 때 평균 실행 완료 시간이 nice 값에 비례한다. 이와 반대로 GPU로 연산했을 경우는 nice 값과 무관한 결과를 보여준다.

사용자는 DNN 작업에 nice 값을 부여하고, CFS는 이를 해당 가중치 값으로 변경하며, 이 변환된 값이 DNN 작업의 QoS 수준을 결정한다. 따라서 CPU 측면에서 CFS가 추구하는 공정 배분 기술이 GPU에 전달되지 않는 한 높은 우선순위를 가진 작업에게 많은 리소스를 할당하여 GPU 점유 시간을 보장하는 QoS에 대한 정확한 성능 격리를 제공할 수 없다.

본 논문에서는 n개의 CPU와 m개의 GPU로 이루어져 있는 서버 GPU 시스템에서 N 개의 DNN 모델을 집합 $DNN=\{D_1, D_2, \cdots, D_N\}$이라 할 때, E_i^{cpu}와 E_i^{gpu}는 D_i 를 CPU, GPU에서 연산했을 경우의 실행 완료 시간을 의미한다. 이때, 성능 격리 메트릭을 식 (4)로 정의한다.

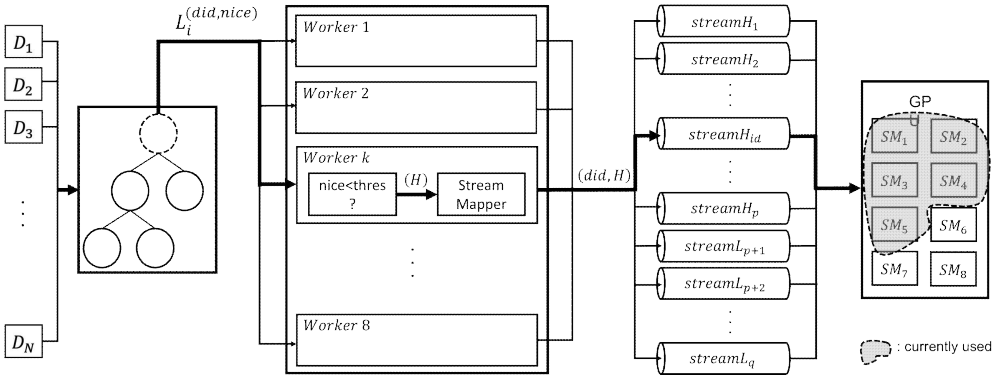
$$\left| \frac{E_i^{cpu}}{E_j^{cpu}} - \frac{E_i^{gpu}}{E_j^{gpu}} \right| = \left| R_i^{cpu} - R_j^{gpu} \right| \tag{4}$$

본 논문에서는 $\{\forall D_i, D_j \mid i \neq j\} \subset DNN$ 일 때, $\min \left| R_{i,j}^{cpu} - R_{i,j}^{gpu} \right|$를 유지하는 것이 목표로 하는 서버 시스템 환경에서 gCFS를 제안한다.

제 3 장 제안 기법

제 1 절 임베디드 시스템용 스케줄링 기법

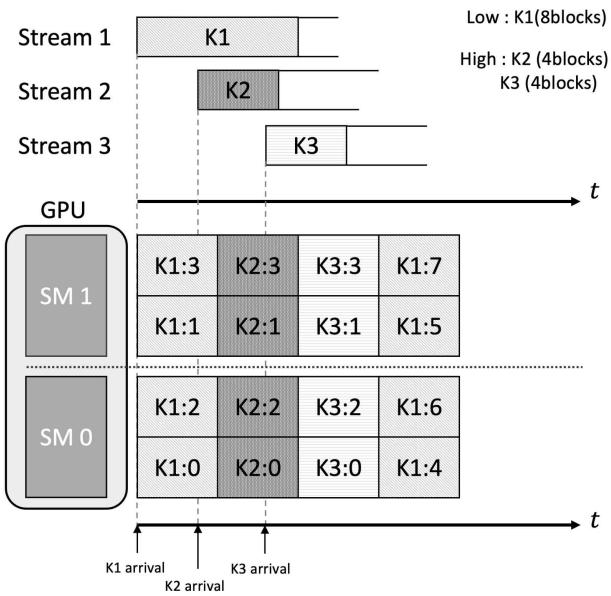
1) 우선순위 기반 다중 DNN 스케줄링 기법의 구조



[그림 3-1] 임베디드 시스템용 우선순위 기반 다중 DNN 스케줄링 기법의 구조

본 논문에서 제안하는 임베디드 시스템용 우선순위 기반 다중 DNN 스케줄링 기법의 구조는 [그림 3-1]에 나타나 있다. 이 기법은 DNN 모델 스레드 집합, 우선순위 큐와 우선순위 스트림 집합, 워커 스레드 풀(pool) 4가지의 구성 요소가 존재한다.

DNN 모델 스레드 집합은 현재 시스템에서 실행되는 N개의 DNN 모델들을 각각 실행하는 스레드들의 집합이다. 각 DNN 모델들은 $D_{did}^{nice} = \{L_1^{(did,nice)} \quad L_2^{(did,nice)}, \cdots, L_j^{(did,nice)}\}$로 나타내며 j는 해당 모델의 총 레이어 개수, $nice$는 CPU 측 CFS의 nice 값을 의미한다. 이때, 하나의 DNN 모델 내부의 모든 레이어는 동일한 $nice$가 부여되며 각각 DNN 모델을 식별할 수 있는 식별자 did $(1 \leq did \leq N)$를 가진다. 각 스레드는 DNN 모델을 레이어 단위로 나누어 우선순위 큐에 하나의 레이어씩 전달하는 역할을 한다.



[그림 3-2] 우선순위 스트림 동작 시 선점이 발생하는 경우 예시

우선순위 큐는 DNN 모델 스레드로부터 전달받은 모든 레이어 연산을 *nice*가 낮은 즉, 우선순위가 높은 레이어가 큐의 top에 위치하게끔 재배열하는 역할을 한다. 우선순위 큐는 항상 큐 내부를 구성하고 있는 레이어 중 가장 우선순위가 높은 레이어를 워커 스레드 풀에 전달한다.

우선순위 스트림 집합은 현재 시스템에서 사용되는 모든 스트림의 집합을 의미한다. 하나의 DNN 모델당 하나의 스트림을 가지며 식별자인 did로 구분한다. 다시 말해, $L_i^{(id,nice)} \in D_{did}^{nice}$ 는 $stream_{id}$에 매핑된다. 각 스트림은 우선순위 값 H,L 중 하나의 우선순위를 가진다. 우선순위 H를 가진 스트림인 $streamH$가 p개, 우선순위 L을 가진 스트림인 $streamL$이 q개인 스트림 집합은$stream = \{streamH_1, \cdots, streamH_p, streamL_{p+1}, \cdots, streamL_q\}$로 정의된다. 이때, $1 \leq p+q \leq m$을 만족한다. 어떤 커널 K1이 $streamL$에 할당되어 먼저 실행되고 있어도 $streamH$에 할당된 커널 K2가 GPU에 전달된다면 K2가 K1을 선점한다.

이때, [그림 3-2]는 위에서 언급한 스트림의 우선순위에 따른 커널 사이에서 선점이 발생하는 예를 보여준다. 두 개의 SM으로 구성된 GPU에서 그림의 $Ki:j$는 커널 i의 j번째 블록을 나타낸다. 현재 스트림 1은 L 우

선순위, 스트림 2, 3은 H 우선순위가 할당되어 있고 $K1$, $K2$, $K3$은 각각 8, 4, 4개의 스레드 블록(thread block)으로 구성되어 있다. 모든 커널의 각 스레드 블록은 같은 스레드 개수로 구성되어 그 크기가 같고, 하나의 SM은 한 번에 두 개의 블록까지 동시에 수행할 수 있다고 가정한다.

$K1$ 커널이 먼저 GPU에 전달되어 $K1$의 4개의 블록(0, 1, 2, 3)이 $SM0$, $SM1$에 분산되어 할당된다. 실행 도중 H 우선순위 스트림의 $K2$가 도착하면 $K1$의 남은 4개의 블록(4, 5, 6, 7)을 H 우선순위 스트림의 $K2$의 4블록(0, 1, 2, 3)이 선점한다. 물론 이때 블록 단위 스케줄링 특성으로, $K1$의 0, 1, 2, 3 블록의 실행 완료 시간보다 $K2$의 도착시각이 빨라도 $K1$의 0, 1, 2, 3블록의 실행이 끝날 때까지 선점이 일어나지 않는다. 이어서 $K2$의 모든 블록의 실행이 완료되는 시점에 $K1$이 또다시 우선순위에서 $K3$에 밀려 $K1$의 남은 4블록(4, 5, 6, 7)은 $K3$가 완료된 후 SM에 할당되어 실행할 수 있다.

워커 스레드 풀은 n개의 CPU에 각각 하나씩 고정(pinning)된 n개의 스레드로 구성된다. 우선순위 큐에서 $L_i^{(did,nice)}$가 전달되면 먼저 우선순위인 $nice$ 값을 확인한 후 $thres$값과 대소를 비교한다. 그 후 *Stream Mapper*를 통하여 적절한 스트림에 $L_i^{(did,nice)}$를 할당한다. 여기서 $thres$값은 $streamH$의 총 개수인 p를 의미한다.

2) 우선순위 기반 다중 DNN 스케줄링 기법의 동작 방법

본 논문에서 제안하는 스케줄링 기법은 먼저 DNN 모델에 $nice$값을 부여한다. 우선순위가 부여된 여러 개의 DNN_{did}^{nice}모델들은 레이어로 순서 즉, $L_1^{(did,nice)}, L_2^{(did,nice)}, \cdots, L_q^{(did,nice)}$ 의 순서로 우선순위 큐에 전송된다. 우선순위 큐는 항상 가장 높은 우선순위의 레이어를 찾아내어 워커 스레드 풀에 전달한다. 워커 스레드 풀은 전달된 $L_i^{(did,nice)}$의 $nice$값과 $thres$값을 비교하여 $nice < thres$인 경우 $streamH_{did}$로, 그 반대의 경우는 $streamL_{did}$에 할당한다. 스트림을 통하여 GPU에 해당 레이어의 연산에 해당하는 커널이 전송되면 GPU는 스트림의 우선순위를 고려하여 각 SM에 적절하게 스케줄링한다.

[그림 3-1]은 전체적인 동작 과정을 예를 들어 설명하고 있다. DNN_{did}^{nice}모

델의 한 레이어인 $L_i^{(did,nice)}$가 현재 우선순위 큐 내부에서 가장 높은 우선순위를 보유하여 워커 스레드 풀로 전송된다. 이 후 $nice < thres$관계를 만족하여 *StreamMapper*는 H 신호를 수신한다. *Stream Mapper*는 $L_i^{(did,nice)}$를 id와 H 신호에 근거하여 H 우선순위를 가진 $streamH_{did}$에 매핑한다. 이때, DNN 모델 집합으로 하여금 다음 수행할 레이어인 $L_{i+1}^{(did,nice)}$를 우선순위 큐에 전송하게 한다. 워커 스레드 풀에서 일어나는 이러한 과정은 비동기식으로 진행된다. 다시 말하면 우선순위 큐의 top에 연산해야 하는 레이어가 존재하고 워커 스레드 풀의 임의의 한 스레드가 유휴(idle) 상태가 될 때마다 top에 위치한 레이어가 이 스레드에 전달된다. 이후, 해당 레이어 $L_i^{(did,nice)}$를 스트림에 매핑한 후에 DNN 모델 스레드 집합에서 $L_{i+1}^{(did,nice)}$가 우선순위 큐에 전송되므로 DNN_{id}^{nice}의 모든 레이어들은 순차적으로 연산 된다.

[그림 3-1]의 예에서는 현재 $L_i^{(did,nice)}$가 $streamH_{did}$에 할당되어 GPU에 전송되는 시점 전에 $SM_1 \sim SM_5$를 다른 커널이 사용하고 있는 상태를 나타낸다. 이때, $SM_1 \sim SM_5$에 L 우선순위 스트림에서 할당된 커널들이 수행되고 있고, 새롭게 입력된 $L_i^{(did,nice)}$의 블록 크기가 $SM_6 \sim SM_8$에서 수용 가능한 크기보다 클 때는 $SM_1 \sim SM_5$에 할당된 커널의 블록 단위 연산이 종료되면 바로 이어서 $SM_1 \sim SM_8$ 중 적절한 숫자의 SM에 할당된다. 반면, $L_i^{(did,nice)}$의 블록 크기가 $SM_6 \sim SM_8$에서 수용 가능한 크기보다 작거나 같을 때는 바로 $SM_6 \sim SM_8$에서 연산을 시작한다. DNN_{did}^{nice}의 마지막 j번째 레이어 $L_j^{(did,prio)}$의 연산이 완료되면 추론(inference) 결과를 출력한다.

제 2 절 서버 시스템용 스케줄링 기법

본 절에서는 서버 시스템용 스케줄링 기법인 gCFS를 제안한다. 그런 다음 CFS에서 공정한 리소스 할당을 위한 기술이 다중 GPU 환경에서도 유사하게 구현될 방법을 설명한다. 본 절에서 제안하는 스케줄링 기법을

이해하기 위해 자주 사용되는 표기법을 [표 3-1]에 나타냈다.

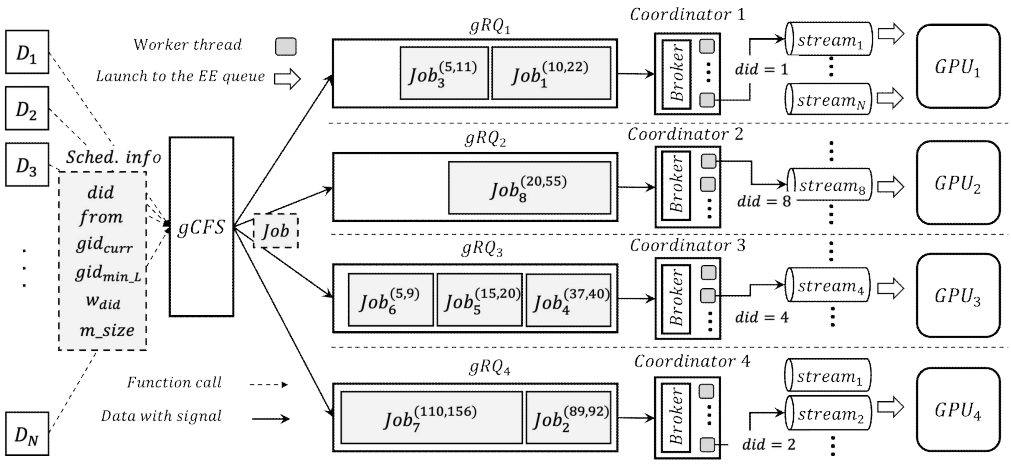
Symbol	Definition
n	Number of CPUs
m	Number of GPUs
gid	GPU index; $gid \in \{1, 2, \cdots, m\}$
N	Number of DNN tasks
did	DNN task index; $did \in \{1, 2, \cdots, N\}$
DNN	Set of DNN tasks $\{D_1, D_2, \cdots, D_N\}$
w_{did}	Weight of DNN task D_{did}
gRQ_{gid}	GPU run-queue with GPU index gid
Job	Some consecutive layers in a DNN model
$Job_{did}^{(from\ ,\ to)}$	Job in D_{did} where the start/end layer index is $from/to$

[표 3-1] 표기법

1) gCFS의 구조

[그림 3-3] 는 gCFS 전반적인 동작 방법을 보여준다. 솔루션 아키텍처는 (1) DNN 모델 집합, 즉 $DNN = \{D_1, D_2, \cdots, D_N\}$ (2) GPU 스케줄러(gCFS), (3) GPU 런큐$(gRQ_1 \sim gRQ_4)$, (4) 코디네이터($Coordinator$)의 네 가지 구성 요소로 구성된다. 이때, 그림 2는 4개의 GPU로 구성된 시스템을 보여주지만, 본 논문의 솔루션은 GPU의 수에 국한되지 않는다.

DNN에 속해있는 작업들은 하나의 DNN 모델을 수행하며 각각의 작업들은 함수 호출을 통해 스케줄링 정보를 비동기적으로 gCFS에 전달 할 수 있다. $D_{did} \in DNN$가 작업 실행을 요청하면 gCFS는 GPU 런큐 중 하나를 선택하여 가장 적절한 GPU를 결정한다. 그 후 gCFS는 Job을 출력하는데, 이는 현재 스케줄링 라운드에서 실행될 D_{did} 의 연속된 레이어 집합을 나타낸다. [그림 3-3]의 $gRQ_1 \sim gRQ_4$에 속해있는 각 Job의 길이는 GPU에서의 실행 시간을 의미한다.



[그림 3-3] 서버 시스템용 gCFS의 구조

GPU 런큐는 전달받은 작업들을 임시로 보관한다. CFS의 런큐는 RB트리[1]로 구성되어있지만 GPU의 런큐는 단순한 FIFO 큐이다. 각 GPU 런큐는 자체 코디네이터에 연결되어 있고, 각 코디네이터에 속해있는 각각의 브로커(*Broker*)는 $Job_{did}^{(from\ ,\ to)}$를 가져와 적절한 CUDA 스트림 [14]에 연결하기 위해 워커 스레드(*Worker thread*) 중 하나에 매핑한다. 각 GPU는 N개의 스트림을 가지며, did는 스트림 번호를 결정한다 (예를 들어, $stream_{did}$는 $Job_{did}^{(from\ ,\ to)}$에 할당된다). 마지막으로 $Job_{did}^{(from\ ,\ to)}$은 GPU의 $stream_{did}$를 통해 EE 큐에 런치(launch)되어 연산들이 실행된다 [13].

여러 워커 스레드에 의해 트리거 되는 여러 스트림을 이용함으로써 SM(streaming multi processor), 레지스터 파일 및 글로벌 버퍼 메모리의 내부 리소스 가용성에 따라 단일 GPU에서도 여러 작업을 병렬로 실행할 수 있다.

본 절에서 제안한 솔루션인 gCFS가 실행되는 과정을 나타낸 수도 코드를 [그림 3-4]에서 설명하고 있다. 여러 DNN 작업은 gCFS를 통해 모델 실행을 동시에 요청할 수 있다. 이때, $SelectQ(\bullet)$ 과정이 첫번째로 실행되며 gRQ_{gid} 에 해당하는 gid를 출력한다. 그런 다음 gCFS는 D_{did}에서 모델의 타임 슬라이스 ts를 계산하고 $Job\_Form(\bullet)$ 함수로부터 $Job_{did}^{(from\ ,\ to)}$을 얻는다.

Algorithm 1 gCFS

Require: $gid_{\min L}$: index of the GPU with the minimum load,
$from$: start layer index of D_{did} at this scheduling round,
m_{size} : output size of the $(from-1)^{th}$ layer in D_{did},
gid_{curr} : GPU index for D_{did} in the previous scheduling round

1: **function** gCFS($did, from, gid_{curr}, gid_{\min L}, m_{size}, w_{did}$)
2: $gid \leftarrow$ SelectQ($gid_{curr}, gid_{minL}, m_{size}$)
3: $ts \leftarrow$ Cal_Timeslice (gid, w_{did})
4: $Job_{did}^{(from,to)} \leftarrow$ Job_Form($ts, from$)
5: InsertQ($gid, Job_{did}^{(from,to)}$)
6: **end function**

[그림 3-4] gCFS 의 동작 알고리즘을 나타낸 수도 코드

계산된 ts는 D_{did}가 현재 스케줄링 라운드에서 GPU_{gid}를 점유할 수 있는 시간을 의미하며, ts가 만료되기 전에 $Job_{did}^{(from\ ,\ to)}$ 이 실행된다.

2) gCFS의 알고리즘과 동작 방법

가) GPU 런큐 선택 알고리즘

GPU 런큐 선택과정을 [그림 3-5]의 $SelectQ(\bullet)$ 에서 상세히 설명한다. 먼저, 새로 들어오는 D_{did}가 각 GPU 런큐 내의 부하를 통해 대기 시간을 추정한다. 이를 위해 이전 스케줄링 라운드 동안 D_{did} 가 실행되었던 GPU의 인덱스를 나타내는 gid_{curr} 와 최소 부하를 갖는 GPU의 인덱스를 나타내는 $gid_{\min L}$ 의 두 가지 GPU의 $Qtime(\bullet)$을 비교하여 작은 값을 가진 GPU 런큐를 선택한다 ([그림 3-5], 라인 2-8). 여기서 부하는 각 GPU 런큐에 속해있는 DNN 작업의 가중치의 합을 의미한다. $Qtime(gid)$는 스케줄링 되는 Job이 gRQ_{gid} 내에서 예상되는 대기시간을 나타낸다. $Qtime(\bullet)$을 구현하기 위해 각 DNN 모델의 모든 레이어의 실행시간을 오프라인에서 프로파일링을 통해 측정한 후 측정 결과에 대해 각 평균값을 사용했다. 예를 들어, [그림 3-3]의 $Qtime(gid=4)$은 $Job_7^{(110,156)}$과 $Job_2^{(89,92)}$의 실행시간의 합과 같으며 각각의 Job의 실행시간은 Job을 구성하는 레이어들의 실행 시간의 평균값 ($layertime(\bullet)$)의 합으로 구한다.

Algorithm 2 GPU selection, Get time slice , Job formation

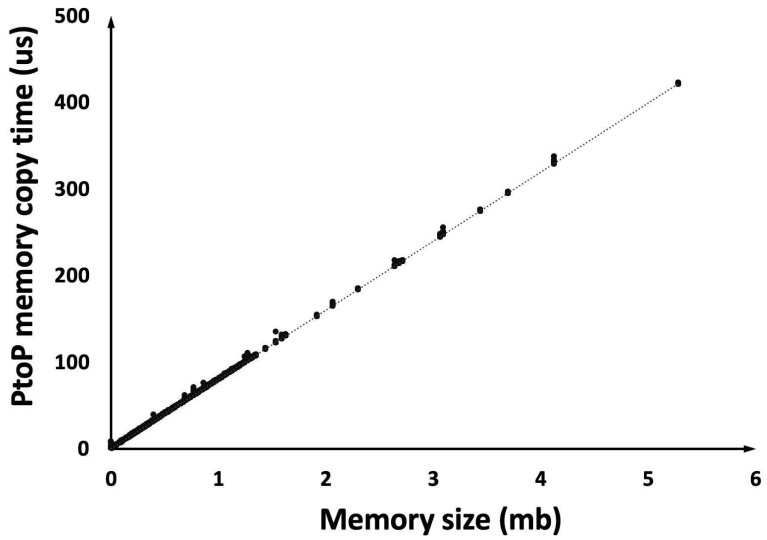
Require: $\mathbb{L} = \{Load_1, Load_2, \cdots, Load_g\}$: set of load for m GPUs

function SelectQ(gid_{curr}, $gid_{\min L}$, m_{size})
  $T_{curr} \leftarrow$ Qtime (gid_{curr})
  $T_{\min L} \leftarrow$ Qtime ($gid_{\min L}$) + copytime(m_{size})
  if $T_{curr} > T_{\min L}$ **then**
    $gid \leftarrow gid_{\min L}$
  else
    $gid \leftarrow gid_{curr}$
  endif
end function
function Cal_Timeslice(gid, w_{did})
  $Load_{gid} \leftarrow Load_{gid} + w_{did}$
  $ts \leftarrow \frac{w_{did}}{Load_{gid}} \times P$ ▷ P: duration of one scheduling round
  return ts
end function
function Job_Form(ts, $from$)
  for $l \leftarrow from$ to idx **do** ▷ idx: last layer index of the DNN with did
    $sum \leftarrow sum$ + layertime(l)
    if $ts == sum$ **then return** $Job_{did}^{(from,l)}$
    else if $ts < sum$ **then return** $Job_{did}^{(from,l-1)}$
    end if
  end for
end function

[그림 3-5] gCFS의 세부적인 동작 수도 코드

GPU 런큐 선택 시 D_{did} 의 이전 스케줄링 라운드의 GPU 인덱스와 현재 스케줄링 라운드의 GPU 인덱스가 다르면 두 개의 서로 다른 GPU 간의 메모리 복사가 필요하며 이는 무시할 수 없다. 이를 위해 gid_{curr}의 GPU와 $gid_{\min L}$의 GPU 사이의 메모리 복사과정에 필요한 시간을 $copytime(\bullet)$으로 알 수 있다. 본 기법에서는 NVIDIA가 지원하는 peer-to-peer 통신 [15]을 이용한 데이터 복사를 통해 CPU와 같은 호스트 측 리소스의 개입으로 유발되는 오버헤드를 제거할 수 있다.

$copytime(\bullet)$은 선형회귀(linear regression) 모델을 사용하여 구현했다. 충분한 데이터를 수집하기 위해 무작위로 선택한 여러 DNN 작업을 실행하여 오프라인에서 다양한 종류의 peer-to-peer 통신 시나리오를 수행했다. 수집된



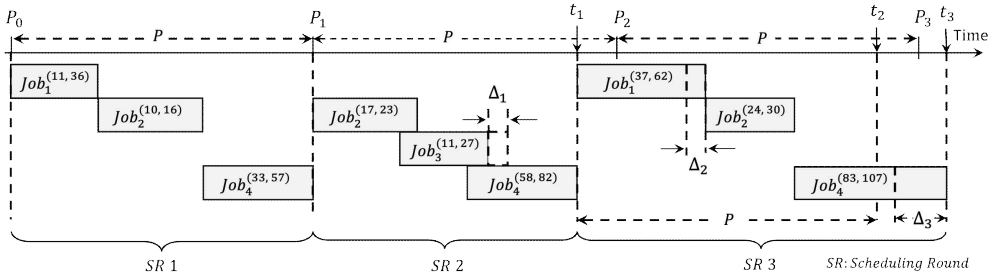
[그림 3-6] 메모리 사이즈별 peer-to-peer 통신을 이용한 데이터 복사 오버헤드

결과를 바탕으로 분석한 결과가 [그림 3-6]에 나타나 있으며, 서로 다른 GPU 간의 메모리 복사에 필요한 시간은 단순히 복사할 데이터 크기에 비례함을 알 수 있다. 따라서 $y = ax + b$ 형태의 간단한 선형회귀로도 충분한 예측 정확도를 얻을 수 있었다. 이때, y는 $copytime(\bullet)$의 결과값이고 x는 복사할 데이터의 크기인 m_{size}이다. 위의 과정들을 통해 얻어낸 T_{curr}과 $T_{\min L}$([그림 3-5], 라인 2-3)을 비교하여 작은 값을 가진 gid를 선택한다([그림 3-5], 라인 4-8).

본 절에서 제안한 기법으로 스케줄링은 기존의 DNN 모델 단위가 아닌 세분화하여 *Job* 단위로 발생하며, 각 스케줄링 라운드에서 가장 작은 부하를 가진 GPU를 고려하여 여러 GPU 간의 워크로드를 균일하게 분산시킨다. 이를 통해 CFS에서 유사한 부하 재분배 효과를 얻을 수 있다.

나) 타임 슬라이스 계산과 *Job* 크기 결정

본 연구의 목적은 서버용 시스템에서 DNN 작업에 주어진 가중치에 따라 GPU 자원을 분배하여 성능 분리를 보장하는 것이다. 이를 위해 CPU에서 사용된 타임 슬라이스의 개념을 GPU에도 적용한다. CFS에서 CPU 점유시간을 얻고자 하는 모든 작업은 식 (1)을 통해 타임 슬라이스를 할당받은 다음 해당 스케줄링



[그림 3-7] gCFS 알고리즘의 동작 과정 예시

라운드동안 타임 슬라이스만큼 CPU를 점유한다. 이처럼 GPU에서도 동일하게 GPU 런큐 내부의 작업들은 각각의 스케줄링 라운드에서 타임 슬라이스를 갖게 한다. [그림 3-5, 라인 11-12]에 설명된 대로 각 스케줄링 라운드에서 각 *Job*의 가중치 즉, 각 DNN 모델의 가중치에 따라 다른 타임 슬라이스가 주어진다. $Load_{gid}$는 인덱스가 gid인 GPU의 GPU 런큐에 속해있는 *Jobs*의 가중치의 합을 의미하며, CFS에서의 부하 계산법과 동일하다.

다) gCFS의 동작 방법

[그림 3-7]에서 4개의 DNN 모델 작업들(D_1, D_2, D_3, D_4)이 각각의 CUDA 스트림을 통해 동일한 GPU에서 실행된다는 간단한 예시를 들어 gCFS의 동작 방법을 설명한다. 그림에서 *Job*의 가로길이는 *Job*에 속해있는 커널들이 GPU에 런치(launch)되는 시간과 SM을 할당받아 연산을 실행하는 시간을 나타낸다. 이론적으로 각 *SR*(scheduling round)의 길이는 [그림 3-5]에 의해 P와 동일해야 하지만, 실제로는 각 GPU 자원 가용성 상태에 따라 $SR = P$, $SR < P$, $SR > P$의 세 가지 경우가 존재 할 수 있다.

SR 1 동안, 세 개의 *Job*이 다른 CUDA 스트림을 통해 GPU에 할당되고, 각 *Job*은 계산된 타임 슬라이스만큼 GPU를 사용한 후 다른 *Job*에게 GPU의 SM을 내어준다. 이는 $SR = P$ 이면서 서로 다른 *Job*들이 GPU의 하드웨어 자원을 동시에 사용하지 않는 경우를 나타낸다.

그 후, *SR* 2는 P_1에서 시작되고 새로운 Job_3이 GPU로 스케줄링 된다. *SR* 2 동안 GPU 리소스 가용성으로 인해 Job_2의 뒷부분과 Job_3의 앞부분이 겹치며, Job_3

의 뒷부분과 Job_4의 앞부분도 마찬가지이다. 이로 인해 Δ_1만큼 Job_3의 실제 실행 시간이 감소한다. 이러한 동시 실행으로 인해 SR 2는 P_1에서 시작하여 P_2에서 종료될 것으로 예상되었으나 t_1에서 조기 종료된다($SR < P$). 이때 스케줄링 라운드 동안 P를 다 사용하지 않았음에도 불구하고, gCFS는 SR 3의 시작을 P_2에서 t_1으로 변경한다.

SR 3에서는 스케줄링 라운드 종료시간을 P_3으로 계산하였으나 예상보다 지연되어 t_3에서 종료되며($SR > P$), Δ_2와 Δ_3은 식 (1)에 의해 계산된 타임 슬라이스보다 지연됨을 나타낸다. 이에 따라, SR 4의 시작은 t_3으로 변경되고 SR 4의 종료시간도 $t_3 + P$ 조정되어 스케줄링 된 Job들이 배치된다.

각 스케줄링 라운드에 대해 계산된 Job들의 타임 슬라이스와 실제 GPU 점유 시간이 일치하면, 각 SR은 P_1, P_2, P_3에 종료되지만 [그림 3-7]과 같이 SR 2처럼 짧을 수도 있고[1], SR 3처럼 Job들의 타임 슬라이스 합보다 길 수도 있다[2].

1) GPU 내부의 SM, shared memory 등의 자원 가용성으로 인해 여러 CUDA 스트림에 전달되는 Job들이 병렬로 처리되는 경우

2) 코디네이터에 속해있는 워커 스레드간의 경합으로 인한 CPU 측의 스케줄링 지연 및 GPU 워프 스케줄러 내부에서 발생한 순간적인 문제로 인해 지연이 발생하는 경우[16]

제 4 장 실험 및 분석

본 절에서는 본 논문에서 제안한 기법의 실용성을 각 시스템에서 입증하기위해 진행한 실험들의 환경과 이들의 결과 및 분석된 내용을 설명한다. 다양한 조건에서 실험을 진행하기 위해 한 종류의 DNN 모델 여러 개를 실행하거나, 서로 다른 종류의 모델을 무작위로 혼합하여 실행하는 방식으로 검증 방법을 다양화하였다.

제 1 절 임베디드 시스템용 스케줄링 기법

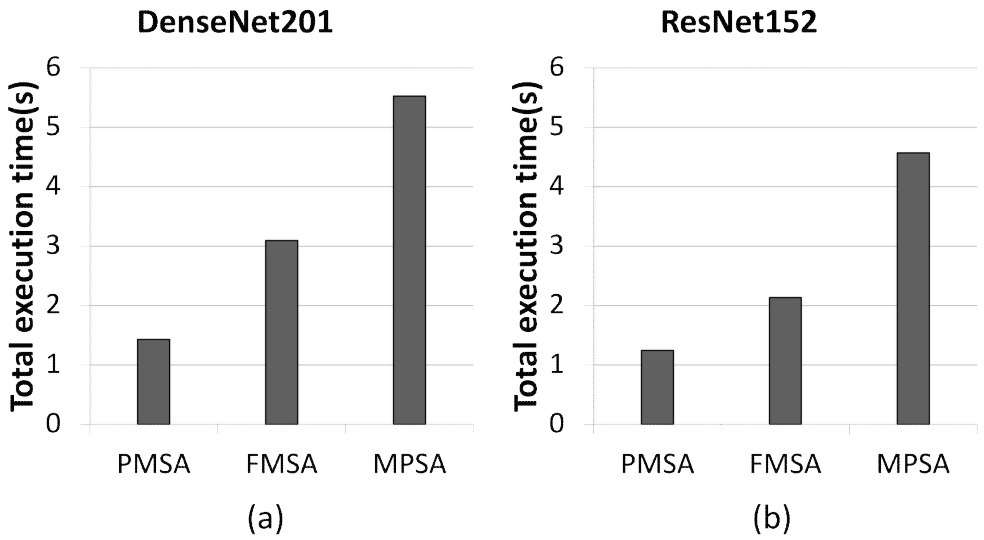
1) 실험 환경

본 논문에서는 NVIDIA(사)의 Jetson AGX Xavier 보드를 임베디드 시스템용 스케줄링 기법을 검증하기 위한 실험의 대상 시스템으로 사용했다. 이는 대표적인 오토노머스 머신용 인공지능 플랫폼이다 [17]. [표 4-1]은 해당 시스템의 구체적인 사양을 나타낸다.

CPU	8-Core ARM v8.2 64-Bit CPU, 8MB L2, 4MB L3 Cache
GPU	512-core Volta GPU with Tensor cores
Memory	16GB 256-bit LPDDR4x 2133MHZ - 137GB/s
Storage	32GB eMMC 5.1
Linux ver.	4.9.140
CUDA ver.	CUDA v10.0.166

[표 4-1] 대상 임베디드 시스템 Jetson AGX Xavier 플랫폼 사양 [17]

본 연구의 실험에서는 제안하는 스케줄링 기법과 범용으로 사용되는 두 가지 대표적인 Multi-DNN 스케줄링 기법을 비교한다. 이들은 각각 우선순위를 고려하지 않고 DNN 모델들의 레이어 연산이 요구되는 순서대로 실행하는 방식과 다수의 딥러닝 프레임워크 (TensorFlow 혹은 Pytorch 등)에서



[그림 4-1] 다수의 동일 종류 DNN 모델 동시 실행 시 우선순위 H인 DNN 모델들의 실행 완료 시간 비교

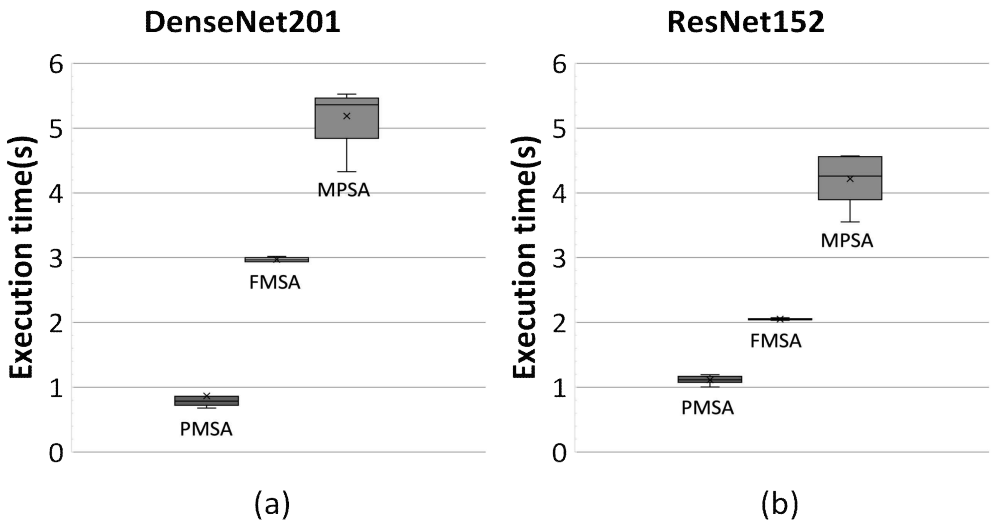
모델을 형성한 후 별도의 프로세스 형태로 구동되는 방식이다. 그리고 이들을 각각 *FMSA*(FIFO-based multi-DNN scheduling algorithm), *MPSA*(m-ultiple DNN process based scheduling algorithm)으로 실험 결과를 나타내고 있는 그래프에 표기하였다. *PMSA*(priority based multi-DNN scheduling algorithm)는 본 연구에서 제안하는 기법을 나타낸다.

실험에서 사용한 DNN 모델은 DenseNet201 [18], ResNet152 [19], VGGNet16 [20], AlexNet [21] 4종류를 사용했으며, 시스템 관리자가 $thres$값을 정의하면 알고리즘 내부에서 자동으로 $streamH$의 개수가 결정되도록 하였다.

2) 실험 결과 및 분석

가) 다수의 동일 DNN 모델 실행

[그림 4-1]는 복수 개의 동일 종류 DNN 모델을 동시에 수행시킬 때 H 우선순위 모델들의 서로 다른 스케줄링 환경 하에서의 수행시간 결과를 비교한다. [그림 4-1]의 (a)는 DenseNet201 30개 중 우선순위가 H인 DNN 모델의 개수가 11개가 되도록 $thres$값을 설정하여 수행한 결과이



[그림 4-2] 다수의 동일 종류 DNN 모델 동시 실행 시 우선순위 H인 DNN 모델들의 실행 완료 시간을 나타낸 박스 플롯 비교

며, (b)는 ResNet152 17개 중 우선순위가 H인 경우가 8개가 되도록 설정한 결과이다. 그림의 (a)를 예를 들면, 우선순위 H인 모델이 11개이므로 $thres$값이 11이 되고, 이때 y축에 나타낸 값은 DNN_1^H ~DNN_{11}^H의 연산이 모두 끝날 때까지 걸린 시간을 의미한다. 그림에서 알 수 있듯이 (a) DensNet201의 경우 FMSA와 비교했을 때 약 53.8%, PMSA와 비교했을 때 약 74.1%의 수행시간이 단축되었다. (b) ResNet152의 경우 FMSA보다 약 41.7%, PMSA보다 약 72.7% 단축된 결과를 볼 수 있다.

[그림 4-2]는 [그림 4-1]과 동일한 환경 하에서 우선순위가 H인 DNN 모델들의 수행시간 분포를 박스 플롯(box plot)을 이용하여 그림으로 나타내었다. H 우선순위를 갖는 모델들의 수행시간 평균값을 비교하면 PMSA가 FMSA, MPSA보다 각각 70.7%, 83.2% 단축되었다. 특히 MPSA와 비교하였을 때 최대 수행시간과 최소 수행시간의 차이는 DenseNet201의 경우 1.1초에서 0.6초로 감소하고 ResNet152의 경우에는 1.02초에서 0.19초로 감소했다.

[그림 4-3]은 H 우선순위 DNN 모델이 L 우선순위 모델을 선점하는 예

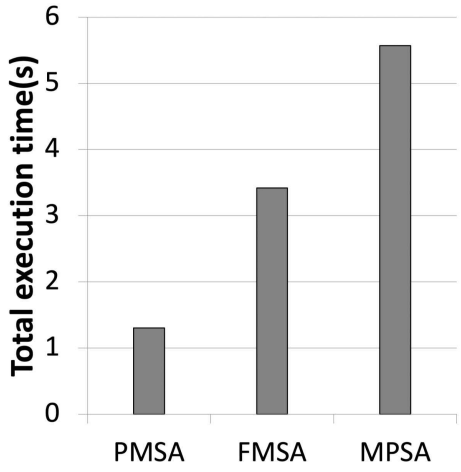
[그림 4-3] NVIDIA Visual Profiler [22]를 사용하여 스트림 H와 스트림 L의 커널을 동시에 실행할 때 선점 효과

를 NVIDIA(사)의 Visual Profiler [22] 프로그램을 통하여 보여주고 있다. 이는 스트림별 커널의 수행시간을 직관적으로 확인할 수 있게 한다. 그림의 *Stream* 33, 34, 35는 우선순위가 H인 스트림이고, *Stream* 36은 우선순위가 L인 스트림이다. 그림의 *kernel start*가 가리키는 시점에서 *Stream* 36의 *forward_maxpool_layer*커널이 먼저 GPU 내부의 SM을 점유한 후 연산을 시작했다. 이 후 그림의 점선 박스 내의 우선순위가 H인 스트림의 커널들이 실행을 요청한다. 이때, *Stream* 36의 *forward_maxpool_layer* 커널은 점선 박스 내의 커널들에 의하여 선점을 당한다. 따라서 *forward_maxpool_ layer* 커널의 완료 시점은 *kernel end* 시점으로 지연된다.

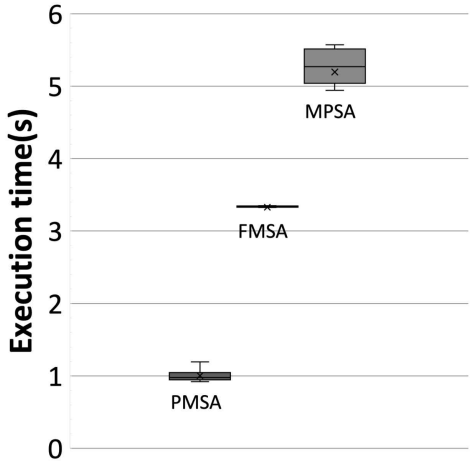
나) 다수의 다중 DNN 모델 실행

다음으로 DenseNet201, ResNet152, VGGNet16, AlexNet 4종류 모델들을 복수 개로 형성한 후 동시에 수행되는 환경에서, H 우선순위를 갖는 모델들이 모두 끝나는 시간과 이들의 수행 시간 분포를 비교하였다. 이때 DenseNet201은 18개, ResNet152 4개, VGGNet16 2개, AlexNet 1개를 사용하여 총 25개의 DNN 모델을 구성하고 이들 중 DenseNet201 모델 9개만 H의 우선순위를 갖도록 설정하였다. [그림 4-4]는 이의 결과를 나타내며 PMSA는 FMSA와 비교 시 61.8%, MPSA와 비교하면 76.6% 감소한 결과를 나타내었다.

[그림 4-5]는 [그림 4-4]와 같은 환경 하에서 수행시간 분포를 박스 플롯(box plot)으로 나타내었다. 그림에서 알 수 있듯이, 중앙값과 전체적인 수행 시간이 PMSA가 가장 작고 H 우선순위를 갖는 모델들의 수행시간 평균값 또한 PMSA가 FMSA, MPSA보다 69.8%, 80.7%로 개선되었다. 특히 MPSA와 비교했을 때 최대 수행시간과 최소 수행시간의 차이가 1.2초에서 0.2초로 향상되었다.



[그림 4-4] 다수의 여러 종류 DNN 모델 동시 실행 시 우선순위 H인 DNN 모델들의 실행 완료 시간 비교



[그림 4-5] 다수의 여러 종류 DNN 모델 동시 실행 시 우선순위 H인 DNN 모델들의 실행 완료 시간을 나타낸 박스 플롯 비교

결과적으로 다수의 동일한 DNN 모델들을 수행하는 환경뿐만 아니라 다양한 종류의 DNN 모델들이 수행되는 환경 모두 제안된 기법을 사용하면 H 우선순위로 지정한 DNN 모델들의 응답특성을 크게 향상시킬 수 있었다. 또한 이들의 수행 시간 분포 결과에서 보이듯이 동일한 H 우선순위 DNN 모델들이 일정한 수행 시간을 나타내었다.

실험을 통하여 알 수 있듯이 여러 종류의 우선순위를 갖는 DNN 모델들이

사용되는 임베디드 환경에서, 본 연구에서 제안하는 우선순위 기반 스케줄링 기법을 적용하면 어떤 DNN 모델들이 H 우선순위를 부여받을 경우 빠른 응답특성과 일정한 수행시간을 보장받을 수 있을 것으로 기대된다.

제 2 절 서버 시스템용 스케줄링 기법

이 절에서는 gCFS의 유효성을 입증하기 위해 수행한 실험에 대해 자세히 설명한다. 먼저 gCFS를 구현한 방법을 설명한 다음, 측정 결과를 트레이드오프(trade-off) 분석과 함께 설명한다.

1) 실험 환경 및 구현

HW	CPU Intel i9-10940X (x 28)	14 cores / 28 threads 256GB Memory 19.25MB L3 Cache Up to 94 GB/s Memory Bandwidth PCI Express 3.0 x 48
	GPU Quadro RTX 6000 (x 4)	4608 CUDA cores 576 Tensor cores 72 RT cores 24GB Memory Up to 672 GB/s Memory Bandwidth 16.3 TFLOPS (FP32) PCI Express 3.0 x 16
SW	OS	Ubuntu 18.04.4
	Linux ver.	5.3.028
	CUDA ver.	CUDA v10.2

[표 4-2] 대상 서버 시스템 사양

이 실험에서 Pytorch 프레임워크 [23]에서 제공하는 DNN 모델을 $DNN = \{D_1, D_2, \cdots, D_N\}$에 활용한다. [그림 3-2]의 4개의 코디네이터 안의 워커 스레드들과 DNN에 속한 모든 DNN 모델들은 단일 프로세스 안에서 독

립적으로 실행되는 스레드들이다. 이러한 멀티 스레딩 구조는 자원 관리 면에서 여러 가지 이점을 제공한다. 모든 스레드들은 동일한 주소 공간에 접근 가능하므로 동기화 프리미티브 및 데이터 공유 방법론을 제공한다 [24].

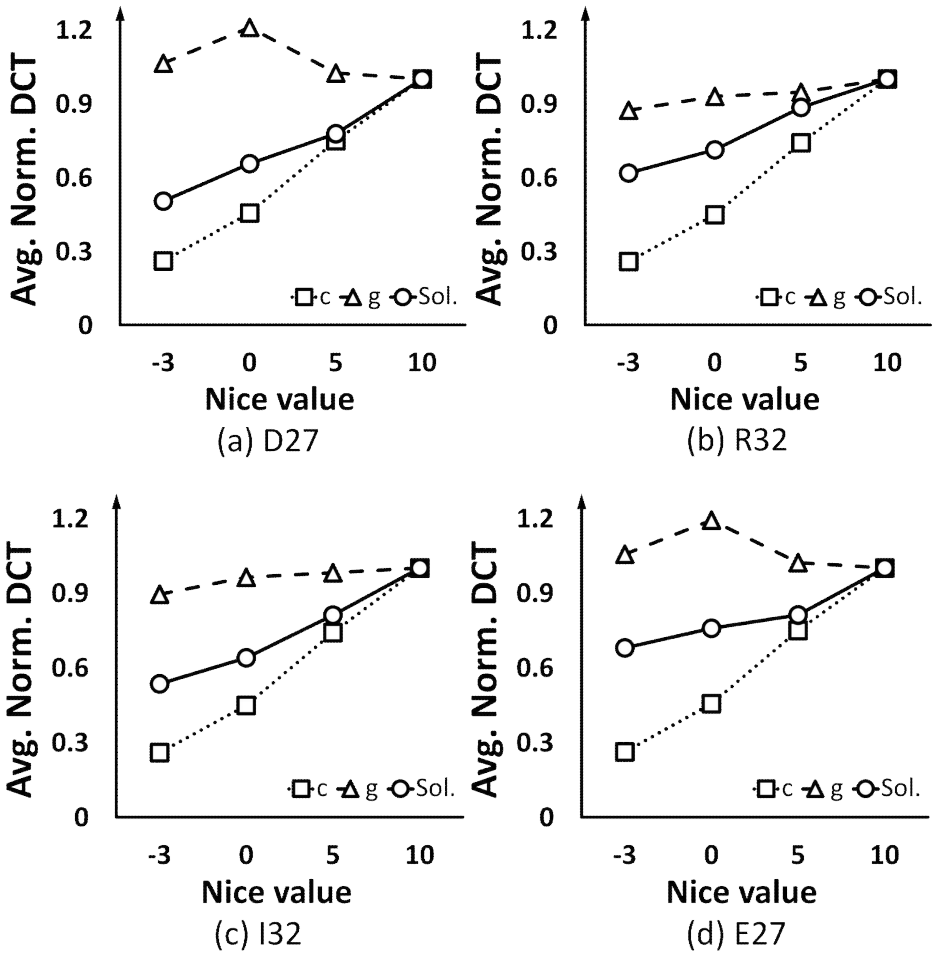
Pytorch의 DNN 모델들은 Python 코드로 구현되어 있으며, 단일 스레드만 Python 인터프리터(interpreter)를 처리할 수 있는 GIL(global interpreter lock)으로 인해 소프트웨어 멀티 스레딩을 적용할 수 없다 [25]. 따라서 본 기법의 멀티 스레딩 기능 구현을 위해 libtorch 라이브러리를 사용하여 Pytorch DNN 모델과 함께 C++ 사용을 가능하게 했다. libtorch 라이브러리를 사용하면 C++ 자체에서 제공하는 Python 보다 빠른 속성과 [그림 3-2]에 나타난 멀티 스레드 기반 스케줄링의 이점을 얻을 수 있다.

gCFS의 실용성을 증명하기 위해 NVIDIA GPU 기반 서버를 대상 시스템으로 정하고 자세한 사양은 [표 4-2]에 나타나 있다. 실험에서 사용한 DNN 모델의 경우, DenseNet201 [18], ResNet152 [19], Inception-v3 [26], EfficientNet b3 [27] 네 종류를 사용했다. 이 절의 모든 그림에서 D27은 27개의 DenseNet201 모델이 동시에 실행 중일 때를 의미하며, D8R8I8E8은 DeneseNet201, ResNet152, Inception-v3, EfficientNet b3이 각각 8개씩 동시에 실행됨을 의미한다.

이 실험에서는 기본적으로 gCFS가 없는 경우와 비교하여 gCFS를 평가하고, CFS가 동작하는 CPU를 통해 DNN 모델 작업이 수행되는 경우를 이상적인 성능 격리가 있는 경우로 설정한다. 각 그림에서 범례 c는 DNN 연산이 전적으로 CPU를 통해 수행되는 경우를 나타내고, g는 gCFS가 없는 모든 GPU에 DNN이 무작위로 분포되었을 때의 결과를 나타내며, $Sol.$ 은 gCFS의 경우이다. 각 실험결과는 연속 100회 이상 수행된 결과의 평균값들이다.

2) 실험 결과 및 분석

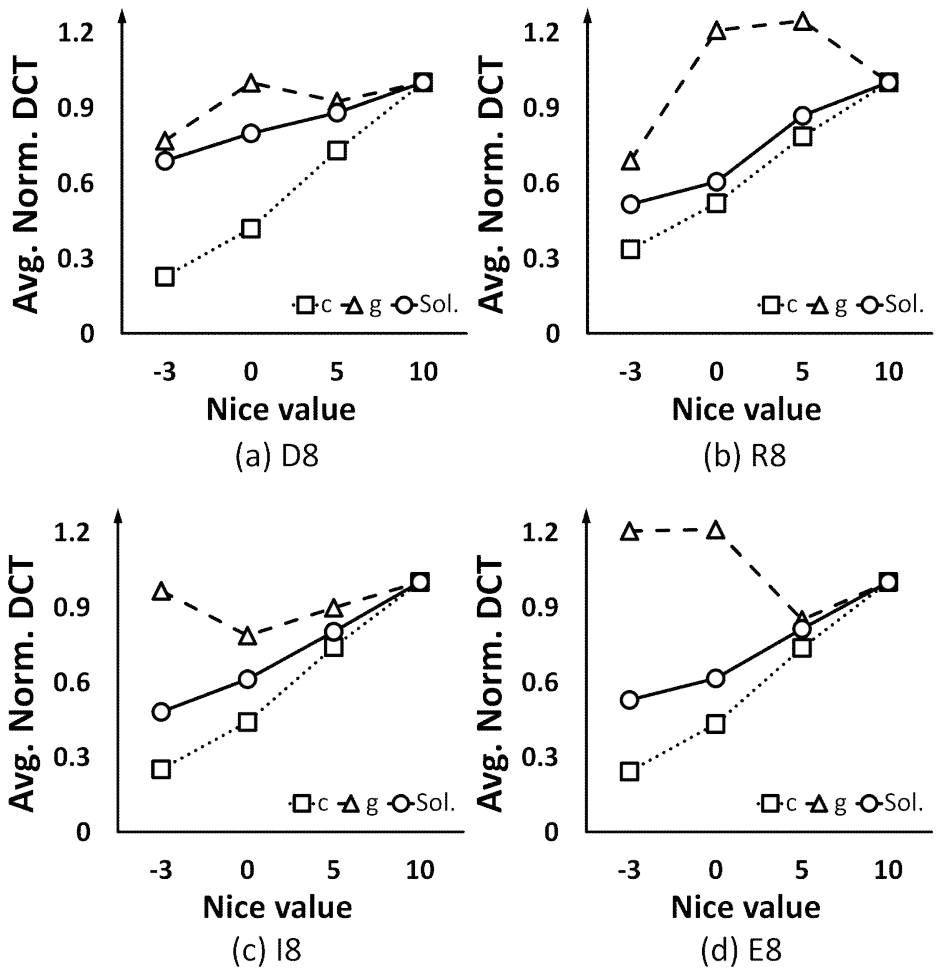
가) 성능 격리



[그림 4-6] 다수의 동일 종류 DNN 모델 동시 실행 시 nice 값에 따른 성능 격리 평가

[그림 4-6]은 여러 개의 단일 종류 모델이 동시 실행 중일때 평균 DCT(DNN model completion time)를 나타내며, 각 실험에서 동일한 모델의 수는 메모리에 의해 제한되며, nice 값 10의 측정값으로 정규화한 결과이다. 모든 DNN 모델의 경우, gCFS가 없을 때 평균 DCT는 주어진 nice 값과 비례하지 않으며, D27의 nice 값 −3에서 이상적인 경우로부터의 최대 80% 차이를 보인다. 하지만 gCFS는 이상적인 경우와의 최대 24% 차이로 우선순위 기반 비례 공정성을 유지한다.

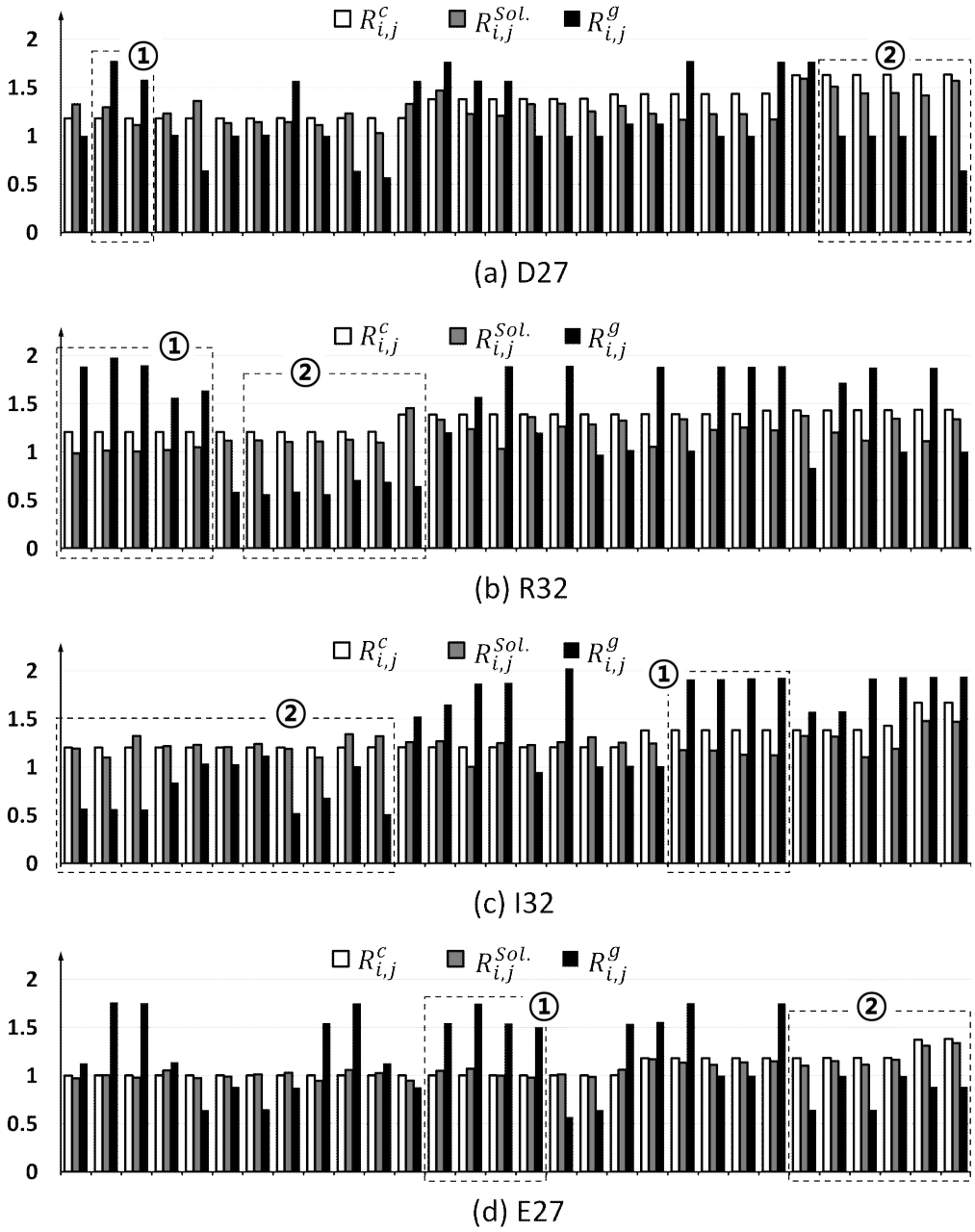
[그림 4-7]은 [그림 4-6]과 달리 서로 다른 모델을 혼합하여 실험하였다.



[그림 4-7] 다수의 여러 종류 DNN 모델 동시 실행 시 nice 값에 따른 성능 격리 평가 (D8R8I8E8)

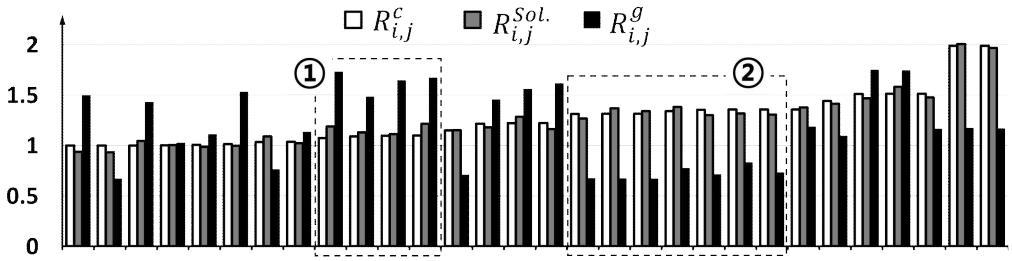
[그림 4-7]은 D8R8I8E8을 실행한 결과를 보여준다. 한 종류의 DNN 모델 당 8개씩 실행되며 동일한 종류의 모델에 −3, 0, 5, 10의 nice 값이 순차적으로 부여된다. 즉, 한 종류의 DNN 모델은 각 nice 값에 대해 2개의 모델을 갖고 있다. [그림 4-7]의 (a),(b),(c),(d)는 동시 실행된 32개의 모델 중 모델 종류 별 결과이다. 다양한 종류의 모델을 동시에 수행하더라도 [그림 4-6]과 유사한 결과를 보여준다. 솔루션을 적용 했을 때 이상적인 경우인 c와 최대 차이가 최대 87.5% 감소함을 알 수 있다.

[그림 4-7]의 (a) 를 보면, 각 nice 값에 따른 성능 격리 정도가 (b), (c),



[그림 4-8] 다수의 동일 종류 DNN 모델 동시 실행 시 성능 격리 메트릭을 사용한 평가

(d)에 비해 나쁨을 알 수 있다. DenseNet201을 구성하는 레이어 수는 다른 3개의 DNN 모델보다 많아 실행 완료 시간이 더 길다. 따라서 다른 DNN 모델은



[그림 4-9] 다수의 여러 종류 DNN 모델 동시 실행 시 성능 격리 메트릭을 사용한 평가 (D8R8I8E8)

추론을 더 일찍 완료할 가능성이 크며, 이때 추론을 완료하지 못한 nice 값이 큰 DenseNet201만 GPU를 독점하여 수행될 가능성이 높다. 이러한 이유로 nice 값이 작은 DenseNet201의 평균 DCT와 nice 값이 큰 평균 DCT의 차이는 다른 DNN 모델의 결과 보다 상대적으로 작을 수 있다.

본 실험에서는 gCFS가 각 DNN 모델에 주어진 nice 값에 맞는 성능을 보장하는지 확인하기 위해 식 (4)에 정의된 성능 격리 메트릭을 사용한다. [그림 4-8]과 [그림 4-9]는 성능 격리 메트릭으로 측정한 결과이며, 각각 [그림 4-6]과 [그림 4-7]에서 수집된 동일한 데이터를 사용한다. 각 그림에서 D_i와 D_j가 mode 실험 환경에서 실행 중일 때, $R_{i,j}^{\mathrm{mode}} = \frac{Avg.\ DCT\ of\ D_i}{Avg.\ DCT\ of\ D_j}$ 이다. D_i와 D_j가 CPU에 의해 실행되면 $\mathrm{mode} = c$, gCFS에 의해 실행되면 $\mathrm{mode} = Sol.$이고, $\mathrm{mode} = g$ 는 gCFS가 없이 DNN 연산을 GPU로 하는 경우이다.

[그림 4-8]은 서로 다른 nice 값을 부여받은 동일 종류의 DNN 모델들이 동시에 실행되었을 때 $R_{i,j}^{\mathrm{mode}}$의 결과이다. [그림 4-8]의 (a), (b), (c), (d)는 $\{\forall D_i, D_j \mid i \neq j\} \subset DNN$, $1 \leq i,j \leq T_{Tot}$를 만족하며 이때, T_{Tot}는 시스템에서 동시에 실행되는 총 DNN 모델 수이고 D_i와 D_j는 동일한 종류의 모델이다. 각 DNN 모델에 대해, $R_{i,j}^{\mathrm{mode}}$의 모든 경우의 수 는 ${}_{N_{Tot}}P_2$이며, 본 실험에서는 ${}_{N_{Tot}}P_2$ 중 30개를 무작위로 나타내었다. [그림 4-9]는 다수의 여러 종류의 DNN 모델 동시 실행 시 $R_{i,j}^{\mathrm{mode}}$의 결과를 보여주며 D_i와 D_j는 이 실험에서 사용한 모든 종류의 DNN 모델을 사용 할 수 있다. [그림 4-8]과 [그림 4-9]의 x축은 무작위로

선택된 $1 \le i,j \le T_{Tot}$, $i \neq j$ 를 만족하는 (D_i, D_j) 쌍 중 무작위로 선택된 30개를 의미한다.

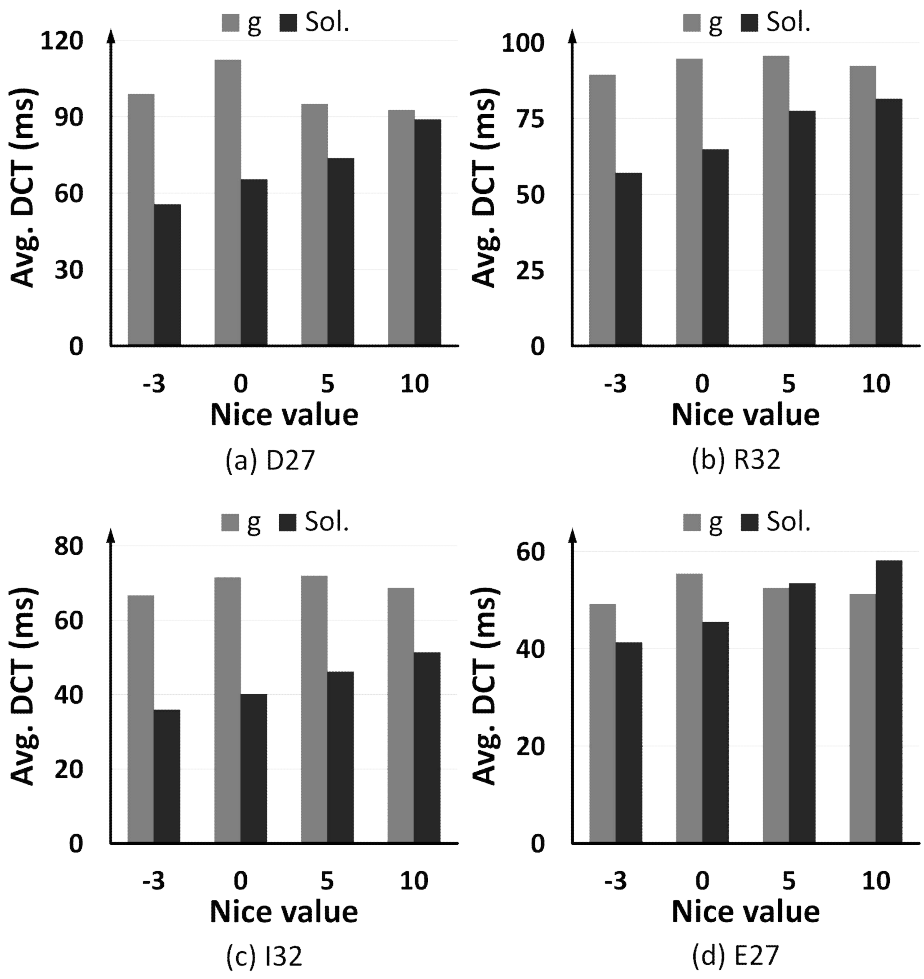
성능 격리 메트릭 측면에서 gCFS와 gCFS가 없는 경우의 직관적 차이를 보다 확실하게 설명하기 위해 [그림 4-8]과 [그림 4-9]에 ①과 ②를 표시하였다. 두 그림에서 ①은 $R_{i,j}^{c}$ 와 $R_{i,j}^{Sol.}$은 유사한 반면, $R_{i,j}^{g}$가 $R_{i,j}^{c}$ 와 $R_{i,j}^{Sol.}$보다 훨씬 큰 경우를 나타내며, ②는 $R_{i,j}^{g}$가 다른 두 값보다 훨씬 작은 경우를 보여준다. 즉, ①과 ②는 $\left|R_{i,g}^{g} - R_{i,g}^{c}\right| \gg \left|R_{i,g}^{Sol.} - R_{c,g}^{c}\right|$를 의미하며 이는 gCFS가 우선순위 기반 성능 격리를 지원함을 뜻한다.

이상적으로는 $\left|R_{i,g}^{Sol.} - R_{c,g}^{c}\right| \approx 0$을 만족해야 하지만 실제로는 다음과 같은 이유로 만족 할 수 없다. CPU의 CFS는 다중 코어 상에서 공정성을 달성하기위해 작업 스케줄링 시 가장 작은 로드를 가진 런큐를 선택하거나 모든 CPU의 런큐에 대해 주기적인 작업 이주(migration)를 발생시키는 두 가지의 부하 재분배를 수행한다. gCFS는 CFS와 달리 새로운 *Job*이 요청될 때 마다 가장 작은 로드를 가진 gRQ_i만 탐색한다는 점에서 CFS의 부하 재분배보다는 미미한 효과를 보일 수밖에 없다. 이는 CFS와 마찬가지로 gCFS가 주기적인 작업 이주를 수행한다면 성능 격리를 더욱 개선할 수 있음을 의미하지만 과도한 peer-to-peer 통신을 이용한 데이터 복사가 수반될 수 있다. 예를 들어 l번째 와 $l+1$번째의 DNN 레이어가 부하 재분배로 인하여 서로 다른 GPU에서 실행될 경우 l번째 레이어의 결과 데이터는 $l+1$번째 레이어의 연산 작업이 할당된 GPU의 메모리에 데이터 복사가 되어야 한다.

따라서 본 연구에서는 최소한의 오버헤드로 $gRQ_1 \sim gRQ_4$에 걸쳐 *Job* 이주를 수행하는 것은 향후 연구 방향으로 설정하였다.

나) DNN 모델 완료 시간(DCT)

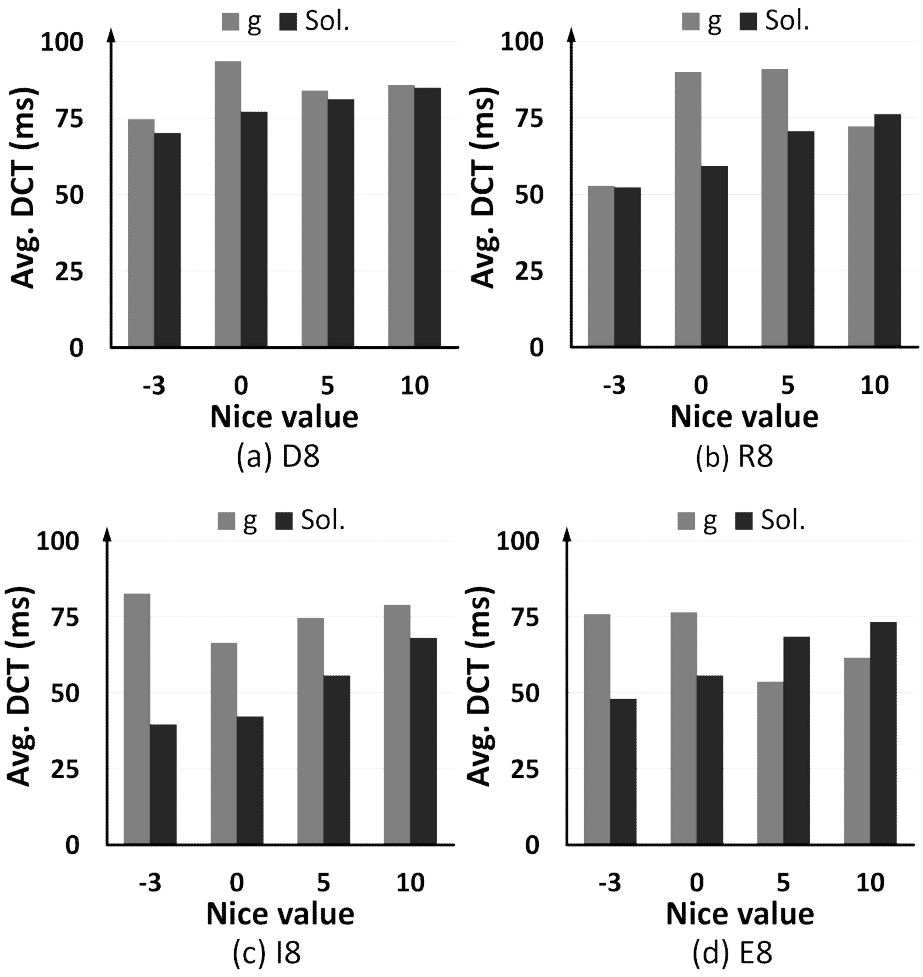
[그림 4-10]과 [그림 4-11]은 각각 동일한 종류의 DNN 모델과 서로 다른 종류의 DNN 모델을 동시에 실행 할 때 nice 값에 따른 DCT의 평균값을 보여준다. 그림에서 보이듯이, DCT의 평균값의 감소는 모든 경우에서 두드러지며, 각 스케줄링 라운드에서 더 큰 타임 슬라이스를 가질 수 있는 작은 nice 값



[그림 4-10] 다수의 동일 종류 DNN 모델 동시 실행 시 nice 값에 따른 DCT의 평균값 비교

(-3,0)에서 최대 41.8%까지 감소하였다. [그림 4-10]의 (d)에서 nice 값이 클 경우 (5,10) gCFS가 없는 결과가 gCFS보다 좋은 결과를 보이지만 이는, g의 경우 nice 값이 작은 모델(-3,0)의 타임 슬라이스를 nice 값이 큰 모델(5,10)에게 뺏겨 우선순위 기반 성능 격리의 목적에 반대된다.

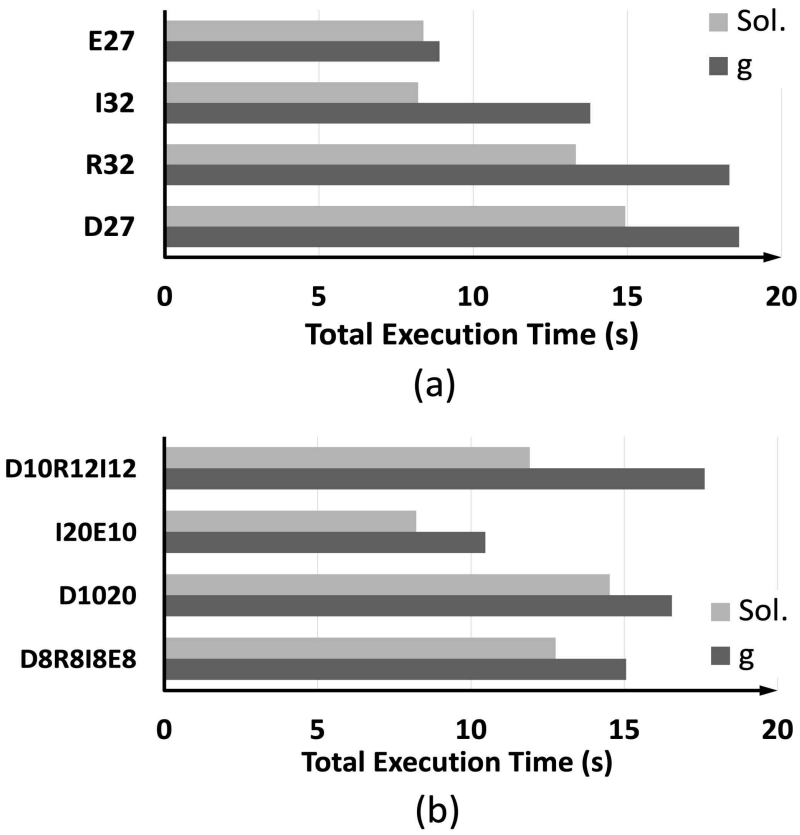
또 다른 메트릭으로 모델 실행 시간 측면에서 gCFS가 효과적이라는 것을 증명하기 위해 모든 작업의 완료시간을 의미하는 makespan을 사용한다. 이를 위해 각 실험의 모든 DNN 모델을 150회 연속 실행 후 완료된 시간을 측정하였다. [그림 4-12]는 다수의 동일한 DNN을 사용했을 때 (a)와 다수의 서로 다른



[그림 4-11] 다수의 여러 종류 DNN 모델 동시 실행 시 nice 값에 따른 DCT의 평균값 비교

DNN으로 구성된 경우 (b)를 나타낸다. 그림에서 알 수 있듯이, 각각의 경우에서 makespan은 최대 40.4%, 32.5%까지 감소한다.

gCFS가 DCT와 makespan에 효과가 있는 이유는 두 가지 요인으로 분석된다. 첫째, DNN 연산을 위한 커널은 기존 DNN 프레임워크처럼 DNN 모델 단위로 GPU에 할당되지 않고 더 작은 *Job* 단위로 할당된다. 더 작은 단위로 큐잉되기 때문에 GPU 런큐에는 항상 *Job*들이 쌓여있다. 따라서 gCFS는 GPU가 유휴(idle) 상태인 시간이 거의 존재하지 않는다. 둘째, gCFS는 멀티 스레딩으로 활성화된 여러 CUDA 스트림을 사용하여 하나의 GPU 내에서라도 SM 가용성에 따라

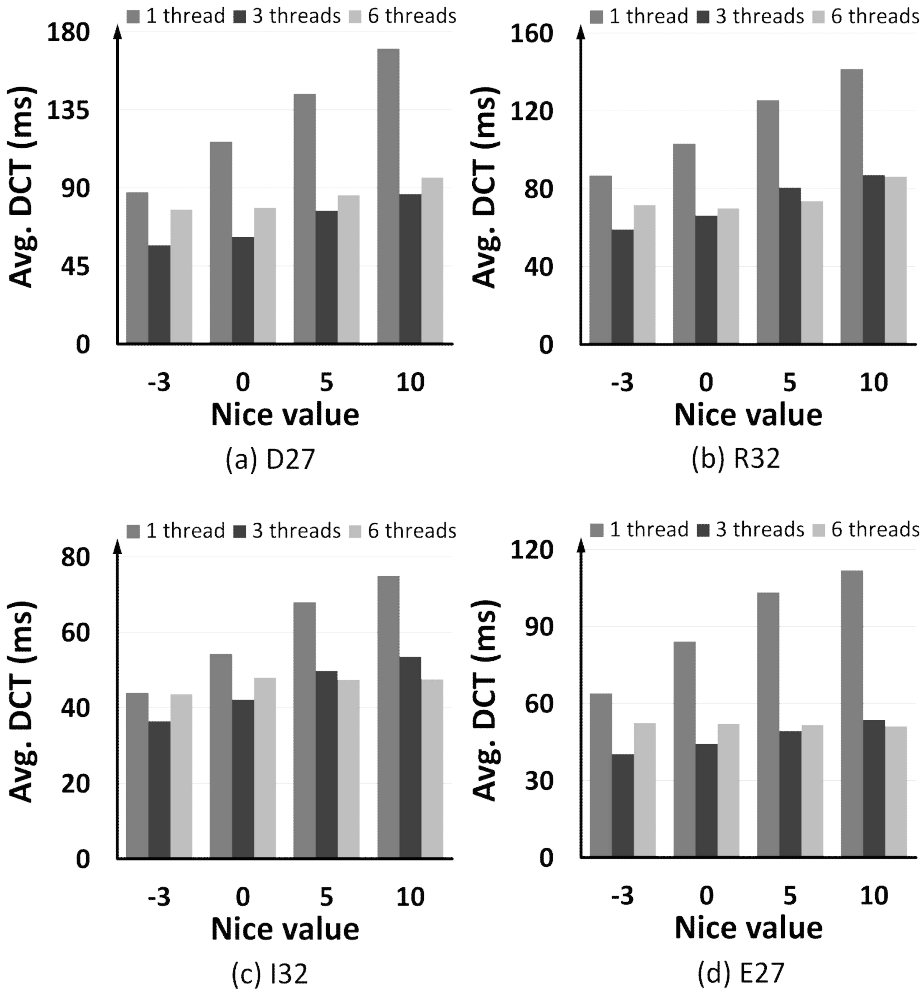


[그림 4-12] 다수의 DNN 모델 동시 실행 시 makespan 비교

여러 연산을 동시에 수행할 수 있다.

3) 트레이드오프 분석

하나의 GPU에 병렬로 동시에 용청할 수 있는 DNN 모델의 수는 GPU당 사용된 스트림 수에 따라 결정되며, 이는 코디네이터당 워커 스레드 수로 정의된다. 따라서 이 숫자는 DCT와 직접적인 관련이 있다. 또한 [그림 3-5]에 표시된 상수 P는 스케줄링 라운드 길이를 나타내며, 이 길이는 해당 스케줄링 라운드에서 어떤 하나의 Job의 스케줄링 여부에 큰 영향을 미친다. 그래서 P는 GPU의 공정한 분배에 중요한 역할을 한다. 이러한 이유로, 본 실험에서는 이 두 가지의 매개변수에 따라 DCT 및 성능 격리를 분석한다.

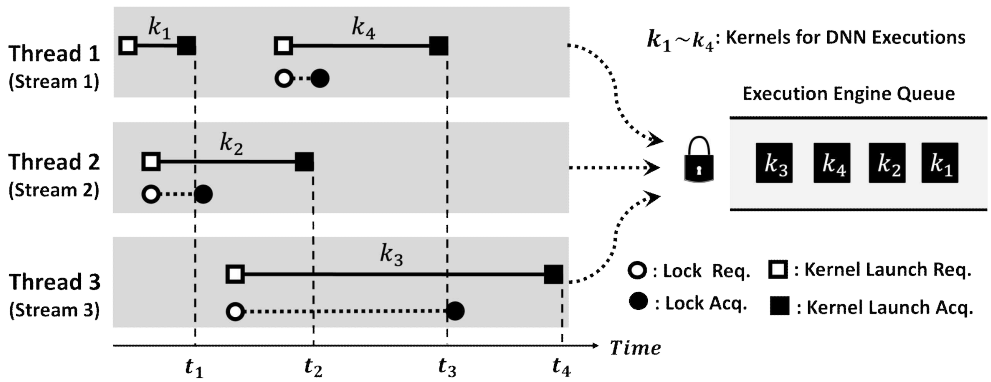


[그림 4-13] 다수의 동일 종류 DNN 모델 동시 실행 시 각 코디네이터의 워커 스레드 수에 따른 DCT의 평균값 비교

가) 코디네이터당 워커 스레드 개수

[그림 4-13]은 각 코디네이터 내부의 워커 스레드 수에 따른 각 nice 값 별 DCT의 평균값을 보여준다. 워커 스레드 개수가 1개, 3개일 때 nice 값이 증가할수록 DCT의 평균값도 증가하는 것을 알 수 있다. 하지만, 워커 스레드 개수가 6개일 때는 nice 값의 증가에 따른 성능 차이가 명확하지 않다.

또한, 높은 우선순위를 나타내는 작은 nice 값 (-3,0)에서 워커 스레드 개수가 6인 경우가 3인 경우보다 더 큰 성능 저하를 보인다. 워커 스레드 수가 많을수록 더 많은 CUDA 스트림을 사용할 수 있으므로 병렬성이 향상되고 성능이 향상



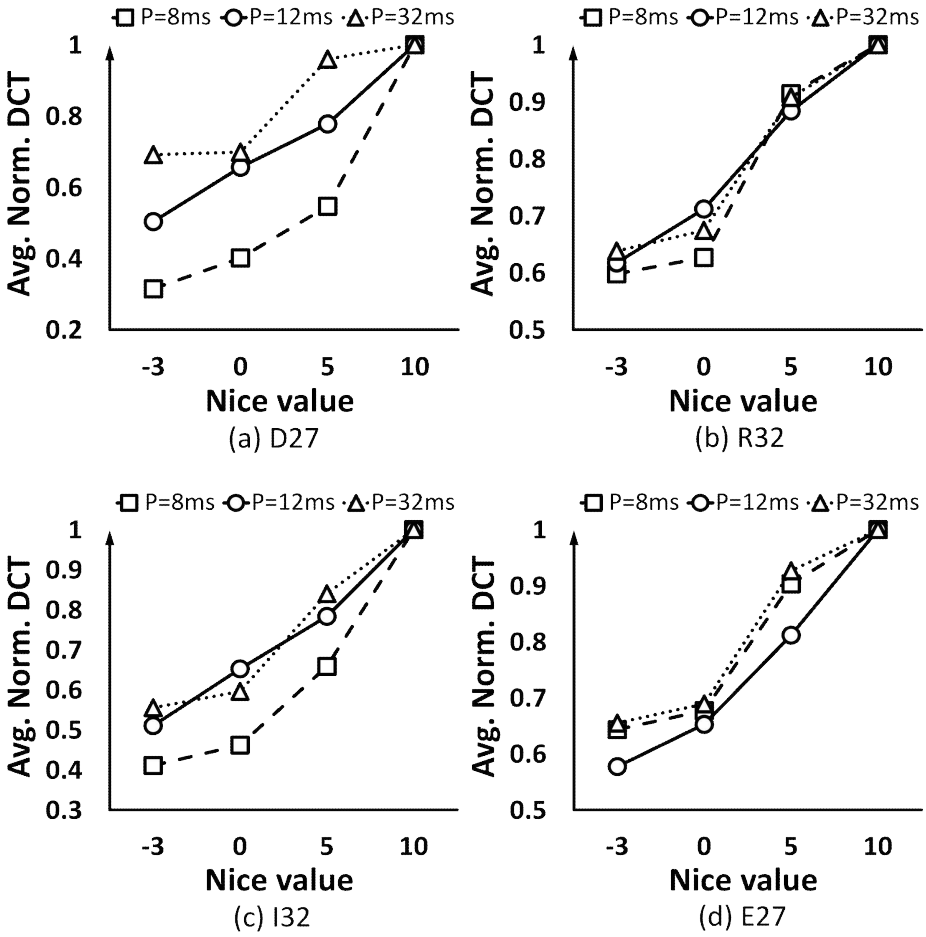
[그림 4-14] 병렬 처리를 위해 여러 스레드를 사용할 때 발생할 수 있는 문제 예시

되어야 한다. 그러나 그림을 통해 알 수 있듯이 실험 결과는 예상과 반대된다. 따라서 NVIDIA에서 제공하는 프로파일링 도구인 Nsight Systems [28]을 사용하여 그 이유를 분석했으며, [그림 4-14]는 이를 설명한다.

각 스레드는 DNN 연산 시 요청되는 커널은 CUDA 스트림을 통해 GPU 당 EE 큐로 비동기적으로 런치 한다. 이때, CUDA API는 한 번에 하나의 커널 실행 요청만 처리할 수 있기 때문에 *pmutex-lock*과 같은 동기화 프리미티브로 인해 직렬화된 작업을 수행한다. 그림에서 $k1$을 실행하려고 할 때 경쟁자가 없기 때문에 즉시 락(lock)을 획득하고 커널을 런치하는 데 필요한 최소 시간이 지나면 $t1$에서 EE 큐로 바로 들어간다. 하지만 $k2$, $k4$, $k3$은 각각 $t1$, $t2$, $t3$에서 락을 획득하고 커널 런치를 시작한다. 또한, 그림에서 $k3$이 $k4$보다 먼저 락을 요청하지만 예기치 않은 CUDA 내부 동작[3]으로 인해 $k4$가 $k3$보다 먼저 락을 획득하여 실행한다는 것을 보여준다. 따라서 멀티 스레드 수가 많을수록 하나의 GPU에 커널을 런치하려는 시도가 많아져 동기화 오버헤드가 발생하고, 락 획득 순서가 커널 런치 요청 순서와 다를 수 있어 잠재적인 문제가 발생할 여지가 있다.

[그림 4-13]과 [그림 4-14]에 나타난 결과를 바탕으로 nice 값에 따른 성능차이를 종합적으로 분석하였다. 또한, GPU 내부의 병렬 처리 과정에서 동기화 문제를 고려한 다음 코디네이터 당 워커 스레드 수를 3개로 결정하였다.

3) CUDA 내부 동작은 공개되어있지 않지만, 프로파일링 결과를 분석해보면 락을 요청한 스레드 중 요청 순서와 상관없이 무작위로 락을 획득하게 하는 것처럼 보임



[그림 4-15] 다수의 동일한 종류의 DNN 모델 동시 실행 시 P 길이에 따른 성능 격리 평가

나) 스케줄링 라운드 P 길이

CFS처럼 gCFS는 식 (1)에 따라 각 *Job*에 타임 슬라이스를 부여하며, [그림 3-5]에 나타낸 바와 같이 타임 슬라이스는 DNN 모델 레이어의 실행 시간 합으로 구성된다. 따라서 P가 너무 짧으면 nice 값이 큰 *Job*은 타임 슬라이스 내에 실행할 수 있는 레이어가 1개보다 작아 스케줄링 될 수 없다. 반면, P가 너무 길면 nice 값에 상관없이 라운드 로빈(round-robin) 형식으로 스케줄링 될 가능성이 높다. [그림 4-15]는 P를 8ms, 12ms, 32ms로 설정하여 측정한 결과이다. 그림에서 보여주듯이, $P=12ms$ 인 경우가 nice 값에 따른 성능 차이를 가장 명확하게 보여준다. 따라서 본 연구에서는 P를 12ms로 설정하였다.

제 5 장 결론

본 논문에서는 CPU 측의 리눅스 CFS의 우선순위(nice 값)가 GPU로 전달되지 못하는 문제점을 GPU 기반 시스템을 임베디드 시스템과 서버 시스템으로 나누어 각 시스템에 맞는 솔루션을 제안하였다.

임베디드 시스템에서는 제한된 환경 내에서 CPU 측면에서 높은 우선순위의 DNN 응용들은 GPU에서 최우선적으로 처리되어야 한다는 목적을 갖고 우선순위 기반 다중 DNN 스케줄러를 제안하였다. DNN 모델들의 레이어들을 실시간으로 입력으로 받아 우선순위 큐로 1차적으로 연산 순서를 결정한다. 이후 CUDA 스트림의 우선순위 기반 선점 기능을 통하여 GPU 내부에서 2차적으로 우선순위가 높은 DNN모델의 레이어가 낮은 우선순위 레이어의 연산보다 먼저 수행되게 한다. 실험 결과 높은 우선순위의 DNN 모델들 중 마지막 모델의 완료시간은 최대 76.6% 감소하였고, 높은 우선순위 모델들의 수행시간 편차 또 한 최대 81.4% 감소하였다.

서버 시스템에서 제안된 gCFS는 CPU 측면에서 전달된 nice 값에 따라 각 DNN이 GPU를 점유할 수 있는 타임 슬라이스 개념을 포함하는 다중 GPU 스케줄러이다. 그렇게 함으로써, gCFS는 각 DNN 모델의 우선순위에 맞춘 격리된 성능을 갖도록 한다. 각 스케줄링 결정 시 gCFS는 가장 적절한 GPU를 동적으로 선택하고 프로파일 기반 DNN 워크로드 배치를 수행한다. gCFS는 이전 연구 작업과 비교하여 세분화된 스케줄링 유닛으로 인해 타임 슬라이스에 대한 보다 정밀한 제어를 지원하여 성능 격리를 향상시켰다. 더 작은 스케줄링 단위를 취함으로써 gCFS는 큐잉 지연이 크게 감소하여 GPU를 더 바쁘게 만들 수 있으며, 이는 또한 DCT 및 makespan 측면에서 성능을 향상시킨다.

향후 연구로 서버 시스템에서 제안하였던 gCFS에서 최소한의 오버헤드로 $gRQ_1 \sim gRQ_4$에 걸쳐 *Job* 이주를 진행 할 계획이며 임베디드 시스템에서도 우선순위 기반의 스케줄링에 더하여 성능 격리를 제공할 예정이다.

참 고 문 헌

1. 국외문헌

Kim, M., Noh, S., Hyeon, J., & Hong, S. (2018). Fair-share scheduling in single-ISA asymmetric multicore architecture via scaled virtual runtime and load redistribution. Journal of Parallel and Distributed Computing, 111, 174-186.

Xiang, Y., & Kim, H. (2019). Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference. In 2019 IEEE Real-Time Systems Symposium (RTSS), 392-405.

Karol, M., Hluchyj, M., & Morgan, S. (1987). Input versus output queueing on a space-division packet switch. IEEE Transactions on communications, 35(12), 1347-1356.

Xiao, W., Bhardwaj, R., Ramjee, R., Sivathanu, M., Kwatra, N., Han, Z., ... & Zhou, L. (2018). Gandiva: Introspective cluster scheduling for deep learning. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), 595-610.

Chen, Q., Yang, H., Mars, J., & Tang, L. (2016). Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. ACM SIGPLAN Notices, 51(4), 681-696.

Chen, Q., Yang, H., Guo, M., Kannan, R. S., Mars, J., & Tang, L. (2017). Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating

Systems, 17-32.

Chaudhary, S., Ramjee, R., Sivathanu, M., Kwatra, N., & Viswanatha, S. (2020, April). Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In Proceedings of the Fifteenth European Conference on Computer Systems, 1-16.

Mahajan, K., Balasubramanian, A., Singhvi, A., Venkataraman, S., Akella, A., Phanishayee, A., & Chawla, S. (2020). Themis: Fair and efficient GPU cluster scheduling. In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), 289-304.

Baruah, S. K., Cohen, N. K., Plaxton, C. G., & Varvel, D. A. (1996). Proportionate progress: A notion of fairness in resource allocation. Algorithmica, 15(6), 600-625.

Jones, M. B., Roşu, D., & Roşu, M. C. (1997). CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. ACM SIGOPS Operating Systems Review, 31(5), 198-211.

Kim, J., Shin, P., Kim, M., & Hong, S. (2020). Memory-aware fair-share scheduling for improved performance isolation in the Linux kernel. IEEE Access, 8, 98874-98886.

Huh, S., Yoo, J., & Hong, S. (2015). Cross-layer resource control and scheduling for improving interactivity in Android. Software: Practice and Experience, 45(11), 1549-1570.

Amert, T., Otterness, N., Yang, M., Anderson, J. H., & Smith, F. D. (2017, December). GPU scheduling on the NVIDIA TX2: Hidden details revealed. In 2017 IEEE Real-Time Systems Symposium (RTSS), 104-115.

Rennich, S. (2012) Cuda c/c++ streams and concurrency. Available online:https://developer.download.nvidia.com/CUDA/training/Strea

msAndConcurrencyWebinar.pdf. Accessed 11 April 2022.

Schroeder, T. C. (2011) Peer-to-peer & unified virtual addressing. Available online: https://developer.download.nvidia.com/CUDA/training/cuda_webinars_GPUDirect_uva.pdf. Accessed 11 April 2022.

NVIDIA (2012) Issue Efficiency. Avaliable online: https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/issueefficiency.htm. Accessed 11 April 2022.

Nvidia Jetson AGX Xavier Developer Kit. Available online: https://developer.nvidia.com/embedded/jetson-agx-xavier-d eveloper-kit.

Yu, X., Zeng, N., Liu, S., & Zhang, Y. D. (2019). Utilization of DenseNet201 for diagnosis of breast abnormality. Machine Vision and Applications, 30(7), 1135-1144.

Nguyen, L. D., Lin, D., Lin, Z., & Cao, J. (2018, May). Deep CNNs for microscopic image classification by exploiting transfer learning and feature concatenation. In 2018 IEEE International Symposium on Circuits and Systems (ISCAS), 1-5.

Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems, 25.

NVIDIA Visual Profiler. Available online: https://developer.nvidia.com/nvidia-visual-profiler.

PyTorch. Available online: https://pytorch.org/. Accessed 11 April 2022.

Lim, C., Kim, M. (2021) ODMDEF: on-device multi-DNN execution framework utilizing adaptive layer-allocation on general purpose cores and accelerators. IEEE Access 9:85403-85417.

Ajitsaria, A.(2020) What is the python global interpreter lock (GIL)?. Available online: https://realpython.com/python-gil/. Accessed 11

April 2022.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., Wojna, Z. (2016) Rethinking the inception architecture for computer vision. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2818-2826.

Tan, M., Le, Q. (2019) Efficientnet: Rethinking model scaling for convolutional neural networks. In: Proceedings of the 36th International Conference on Machine Learning, 6105-6114.

Nvidia nsight systems. Available online: https://developer.nvidia.com/nsight-systems. Accessed 11 April 2022.

ABSTRACT

Priority-bases Performance Isolation and Scheduling in GPU-based Systems for Deep Neural Networks

Cho, Ho-Jin

Major in IT Convergence Engineering

Dept. of IT Convergence Engineering

The Graduate School

Hansung University

In GPU-based systems, CPU applications deliver DNN operations to the GPU. In this case, the DNN operation is transmitted to the GPU in the FIFO method regardless of the priority of the application set by the CPU side OS. In this paper, we propose GPU scheduling techniques for each system by dividing GPU-based systems into embedded systems and server systems. In embedded systems with constrained environments, DNN applications with the highest priority on the CPU side should be handled first on the GPU. In this paper, we propose a scheduling technique that allocates GPU resources considering priorities when multiple DNN models are executed simultaneously. Even if low priority DNN applications are running first, high priority DNN applications can preempt them, ensuring fast response characteristics of high-priority DNN

applications. In a server system composed of multiple GPUs, we propose gCFS allowing each DNN to have GPU occupancy in proportion to the priority. gCFS inherits fair-sharing scheduling on the CPU side and achieves performance separation in proportion to priority. Smaller scheduling granularity enables more precise control over the time slice on GPUs and makes DNN workloads more densely queued, reducing the GPU idle time. in scheduling, the length of DNN workload is adjusted to the given time slice, and dynamically the optimal GPU selection is performed. Through experiments with concurrent running multiple DNNs on each system, the performance improvement is significantly improved compared to the case without scheduling techniques do, and the overall execution time is reduced by up to 76.6% and 40.4%, respectively.

【KEYWORD】 multi-DNN, priority, gpu sharing, performance isolation