석사학위논문

클라우드 비 의존적인 엣지 디바이스 내 최적화된 심층 신경망 연산 기법 연구

2023년

한 성 대 학 교 대 학 원

I T 융 합 공 학 과

I T 융 합 공 학 전 공

임 철 순

석사학위논문
지도교수 김명선

클라우드 비 의존적인 엣지 디바이스 내 최적화된 심층 신경망 연산 기법 연구

Research on Optimized Computation Technique for Deep Neural Network in Cloud-Independent Edge Devices

2022년 12월 일

한 성 대 학 교 대 학 원

I T 융 합 공 학 과

I T 융 합 공 학 전 공

임 철 순

석 사 학 위 논 문
지도교수 김명선

클라우드 비 의존적인 엣지 디바이스 내 최적화된 심층 신경망 연산 기법 연구

Research on Optimized Computation Technique for Deep Neural Network in Cloud-Independent Edge Devices

위 논문을 공학 석사학위 논문으로 제출함

2022년 12월 일

한 성 대 학 교 대 학 원

I T 융 합 공 학 과

I T 융 합 공 학 전 공

임 철 순

임철순의 공학 석사학위 논문을 인준함

2022년 12월 일

심사위원장 오 희 석 (인)

심 사 위 원 이 웅 희 (인)

심 사 위 원 김 명 선 (인)

국 문 초 록

클라우드 비 의존적인 엣지 디바이스 내 최적화된 심층 신경망 연산 기법 연구

한 성 대 학 교 대 학 원
I T 융 합 공 학 과
I T 융 합 공 학 전 공
임 철 순

엣지 디바이스에서 DNN을 효율적으로 연산하는 방법에 대해서는 많은 연구가 이루어지고 있다. 특히 자율주행 분야에서는 네트워크 지연시간과 보안문제로 인해 클라우드 기반 시스템이 아닌 자체 임베디드 시스템에서 처리하는 것이 선호된다. 자율주행 분야에서는 많은 센서 데이터를 처리하기 위해 사용되는 DNN의 수가 늘어나고 있다. 본 논문에서는 DNN연산을 최적화하기 위해 두 가지 관점에서 DNN을 분석한다. 그리고 두 가지 관점의 최적화 기법을 제안한다. 첫번째로 엣지 디바이스에 최적화된 DNN을 탐색하기 위해 NAS시스템을 적용한다. 모바일 환경을 위한 NAS인 MnasNet을 바탕으로 정확도와 실행시간에 대한 최적화를 하는 다중목적 보상함수에 집중한다. MnasNet은 추론 실행시간 제약 조건을 만족하면서 정확도가 가장 높은 DNN을 탐색하는 것이 목표이다. 그리고 주어진 제한시간보다 짧은 실행시간이면서 정확도가 최대가 되는 DNN을 탐색할 수 있도록 새로운 보상함수를 제안한다. 두번째로 여러 개의 DNN을 동시에 실행하였을 때 GPU뿐만 아니

라 CPU도 적절히 이용하는 CPU-GPU Co-Scheduling 프레임워크를 제안한다. 이를 위해 멀티 컨텍스트 환경과 싱글 컨텍스트 환경에서 여러 개의 DNN을 실행될 때 각 환경에서 발생하는 오버헤드를 분석한다. 그 다음 GPU의 계산 부하를 줄이기 위해 사용되지 않는 CPU사이클을 이용한다. CPU와 GPU의 원활한 통신을 위해 오버헤드가 작은 데이터 동기화 방법을 제안한다. 마지막으로 레이어 작업을 실행할 코어를 선택하기 위해 실행시간 예측 모델을 만든다. 이러한 솔루션들을 적용한 후 다중 DNN 실행은 크게 개선되었고 GPU-only 솔루션에 비해 실행시간을 최대 46.6%까지 단축시켰다.

【주요어】 다중 DNN, NAS, GPU 컴퓨팅, 병렬 컴퓨팅, 이기종 컴퓨팅

목 차

그 림 목 차

표 목 차

제 1 장 서론

오늘날 심층신경망(DNN – deep neural networks)들은 로보틱스(Meier, B. J., et al, 2004), 자율주행(Bojarski, M., et al, 2016; Lin, S.-C., et al, 2018) 스마트팜(Vasisht, D., et al, 2017), 광고분야(Cheng, H. T., et al, 2016; He, X., et al, 2014) 등에서 많은 성과를 보여왔다. 특히 DNN들이 오브젝트 탐지 및 분석 분야에서 뛰어난 성능을 보여주었기 때문에 DNN기술들은 자율주행 응용 프로그램 분야에서 많은 연구가 진행되고 있다. 이러한 자율주행 응용 프로그램들은 많은 센서에서 입력되는 데이터를 처리하기 위해 여러 DNN을 사용한다. DNN의 빠른 연산 처리를 위하여 클라우드 컴퓨팅 서버를 사용할 수 있지만 네트워크 지연시간 및 보안문제 때문에 차량 내부 시스템에서 직접 DNN모델을 처리하는 것이 선호된다. 또한 더 정확한 성능을 위해 많은 센서와 많은 DNN을 사용하게 된다. 그래서 엣지 디바이스에서 다중 DNN모델 실행을 위한 다양한 연구들이 이루어지고 있다.

기존에 존재하는 DNN 연산 실행을 최적화 할 수도 있지만, 특정 목적에 맞는 DNN 구조를 직접 생성하거나 변경하는 방법도 있다. 기존에 높은 성능을 보여주었던 Resnet(He, K., et al, 2016)은 잔차학습 구조를 이용하였고, Densenet(Yu, X., et al, 2019)은 Dense블럭을 이용하여 깊은 망에서의 정보 손실을 줄이도록 디자인되었다. 높은 정확도만을 추구하는 DNN들과 달리 클라우드 기반 서버 환경 대비 적은 처리량을 가진 모바일 및 임베디드 시스템에서도 원활하게 실행할 수 있는 DNN에 대해서도 많은 연구가 있다. 경량화된 DNN들의 특징은 시간당 처리량이 작은 환경에서도 처리 속도를 보장하기 위해 정확도가 줄어들더라도 연산량을 줄이도록 디자인 된다. MobileNet(Howard, A. G., et al, 2017)은 연산량이 큰 일반 컨볼루션 레이어 대신 depthwise-seperable 컨볼루션 레이어를 이용하여 전체 연산량을 줄일 수 있었고, ShuffleNet(Zops, B. & Le, Q. V., 2017)는 MobileNet의 depthwise-seperable 컨볼루션 레이어에서 사용하는 1x1 컨볼루션 연산을 연산량이 더 작은 1x1 그룹 컨볼루션 연산으로 대체한다.

언급된 DNN들의 특징은 특정 목적을 위해 사람이 직접 디자인한 DNN 구조라는 것이다. 새로운 DNN 구조를 디자인 하는 것은 많은 시간과 노력을 필요로 한다. 이러한 한계를 벗어나기 위해 NAS(Neural Architecture Search)라는 연구분야가 등장하였다. NAS는 참고문헌(Zhang, X., et al, 2018)에서 처음 제안된 방법이며 컨트롤러라 불리는 RNN모델을 이용하여 DNN 구조를 생성한다. 컨트롤러는 매 반복마다 DNN 구조를 생성하며, 생성된 DNN은 ImageNet(Deng, J., et al, 2009), CIFAR10(Krizhevsky, A. & Hinton, G., 2009)과 같은 데이터셋을 이용하여 정확도를 평가한다. 이 정확도를 이용하여 컨트롤러를 학습시키게 되며 학습이 될 수록 컨트롤러는 더 높은 정확도의 DNN 구조를 생성하게 된다. NAS를 이용하면 직접 DNN 구조를 생성하기 위한 수고를 덜 수 있다.

이러한 NAS를 모바일, 임베디드와 같은 엣지 디바이스에 적용하기 위한 연구도 진행되었다. MnasNet(Tan, M., et al, 2019)은 모바일 환경에서 실행할 수 있는 DNN을 찾기 위한 NAS이다. 앞서 언급한 MobileNet과 SuffleNet과 달리 MnasNet에서는 실행시간에 대한 제약조건을 이용하여 컨트롤러가 생성하는 DNN의 추론 실행시간이 조건보다 낮으면서 높은 정확도를 가지는 DNN을 생성하도록 유도한다. 이렇게 특정 디바이스에서 원활하게 실행될 수 있는 DNN 구조를 탐색하기 위해서 NAS를 적용할 수 있으며 추론 실행시간과 더불어 MAC, 사용하는 메모리와 같은 조건을 이용하여 더 적합한 DNN을 탐색할 수 있다.

NVIDIA(사)의 Jetson AGX Xavier Developer Kit[1]은 자율 주행 응용프로그램과 같은 DNN기반 응용프로그램의 성능을 위해 개발된 고성능 임베디드 시스템이다. 이러한 플랫폼은 CPU와 같은 범용적인 코어와 GPU와 DNN 실행을 가속하기 위한 GPU코어가 존재한다. NVIDIA(사)에서는 자사의 GPU를 위해 CUDA runtime[2]과 cuDNN라이브러리와[3] 같은 소프트웨어를 제공하며, 이는 DNN레이어 연산에 필요한 GPU의 병렬처리 기능을 온전히 활용

1) Nvidia Jetson AGX Xavier,
online: https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit

2) Nvidia Cuda-Toolkit, online: https://developer.nvidia.com/cuda-toolkit

3) Nvidia Cudnn, online: https://developer.nvidia.com/cudnn

할 수 있도록 도와준다.

여러 개의 DNN모델이 GPU에서 동시에 실행될 경우 피할 수 없는 성능 저하가 발생한다. Xavier SoC(system on chip)의 GPU 코어들이 빠른 연산 처리를 위한 Streaming multiprocssors(SMs)와 Hyper-Q 기술[4]을 통한 멀티 쓰레드 실행을 지원하지만, 많은 DNN 모델에서 요청된 GPU 연산은 SMs이 처리할 수 있는 연산량을 넘어서기 쉽다. 그리고 DNN 모델이 별개의 프로세스에서 실행될 경우 컨텍스트 교환 오버헤드가 존재하며, 반면에 한 프로세스 내부에 실행될 DNN들을 쓰레드로 배치하였을 때에는 GPU를 공유자원으로 정의하여 각 DNN 쓰레드마다 동기화 기법으로 제어되기 때문에 이로 인한 오버헤드가 발생한다.

그리고 GPU의 가해지는 부하를 줄이기 위해 일반적으로 DNN실행에 사용되지 않는 CPU를 사용할 수 있다. CPU와 GPU의 명령어 처리 방식, 메모리 구조 등의 아키텍쳐가 다르기 때문에 두 코어는 다른 성능을 보여준다. 이로 인해 CPU-GPU Co-Scheduling을 이용한 DNN실행은 GPU만을 이용하는 것 보다 고려할 문제점이 많으며 오히려 더 낮은 성능을 보여줄 수도 있다. 그리고 DNN 모델 내부의 서로 다른 레이어들은 두 코어간 큰 성능 격차를 보여주기 때문에 오프라인에서 미리 DNN 레이어의 데이터를 수집하여 더 적합한 코어를 선택할 수 있다. 이를 통해 DNN 모델 또는 레이어를 적합한 코어에 할당할 수 있다.

본 논문에서는 엣지 디바이스에서 DNN 모델의 연산을 최적화할 수 있는 방법을 두가지 관점으로 바라본다. 첫 번째는 엣지 디바이스에서 원활하게 실행할 수 있는 DNN을 탐색하는 것이며 이 DNN은 실행시간에 대한 제한조건을 만족하면서 높은 정확도를 갖는 DNN이다. 이를 위해 모바일 환경을 위한 NAS연구인 MnasNet을 기반으로 새로운 다중목적 보상함수를 제안한다. 제안하는 보상함수는 실행시간에 대한 제한조건을 넘는 경우에는 더 낮은 보상을, 제한조건을 만족하는 경우에는 더 큰 보상을 주도록 설계되었다. 이를 통해 정해놓은 제한조건보다 작은 추론 실행시간을 가지면서 높은 정확도

4) Nvidia Hyper-Q, online: http://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf

를 가진 DNN을 더 빠르게 탐색할 수 있게 된다.

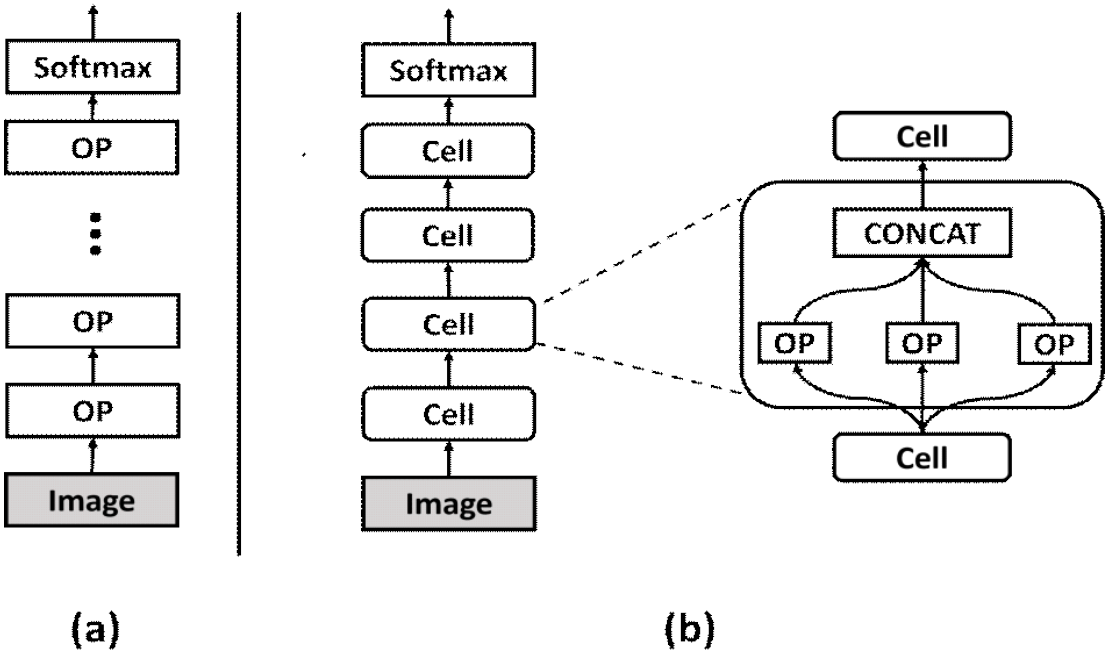
두 번째는 기존에 존재하는 여러 종류의 DNN을 동시에 실행할 때 적절한 스케쥴링 알고리즘을 통해 최적화된 DNN 연산을 제공하는 것이다. 이를 위해 GPU에서의 다중 DNN 실행에서 생기는 문제점을 분석하고 GPU의 병렬처리 성능을 최대한으로 이끌어 내는 방법을 제안한다. 그리고 GPU의 부하를 줄이기 위해 CPU를 이용하는 CPU-GPU Co-Scheduling 프레임워크를 제안한다. 이 프레임워크는 DNN을 레이어 단위로 분리하여 CPU와 GPU에 적절하게 할당할 수 있도록 실행시간 예측모델을 이용하며, CPU-GPU간의 데이터 통신 오버헤드 및 동기화의 오버헤드를 최소화 한다. 현재 연구에서는 DNN 가속기로 GPU를 이용하지만 이 설계는 NPU와 같은 다른 종류의 DNN 가속기에서도 쉽게 적용될 수 있다.

본 논문은 2장에서 연구에 필요한 배경지식에 대한 내용을 서술한다. 그리고 3장에서는 엣지 디바이스를 위한 DNN을 탐색하는 NAS를 제안하고, 4장에서는 다중 DNN을 실행하기 위한 최적화된 프레임워크를 제안한다. 마지막으로 실험을 통해 제안되는 방법들에 대하여 실험 결과를 확인한다.

제 2 장 연구 배경 및 대상시스템 소개

본 장에서는 본 연구의 배경지식인 NAS와 GPU의 동작방식, 타겟 시스템인 Nvidia Jetson AGX Xavier, DNN실행 기법에 대해 분석한다.

제 1 절 NAS(Neural Architecture Search)



[그림 2-1] 레이어 기반 NAS의 DNN(a)와 셀 기반 NAS의 DNN(b)

NAS(neural architecture search)가 등장하기 전의 DNN들은 사람이 직접 디자인한 구조였다. 새로운 DNN을 만들기 위해서는 많은 시간과 노력이 필요하다. 이를 해결하기 위해 강화학습, 진화알고리즘 등을 이용하여 DNN을 디자인하는 NAS가 등장하였다. 이 개념이 처음 등장한 참고문헌 (Zops, B. & Le, Q. V., 2017)에서는 RNN을 이용한 강화학습으로 DNN을 생성한다. 일반적으로 NAS는 DNN을 생성하는 컨트롤러가 있으며, 참고문헌 (Zops,

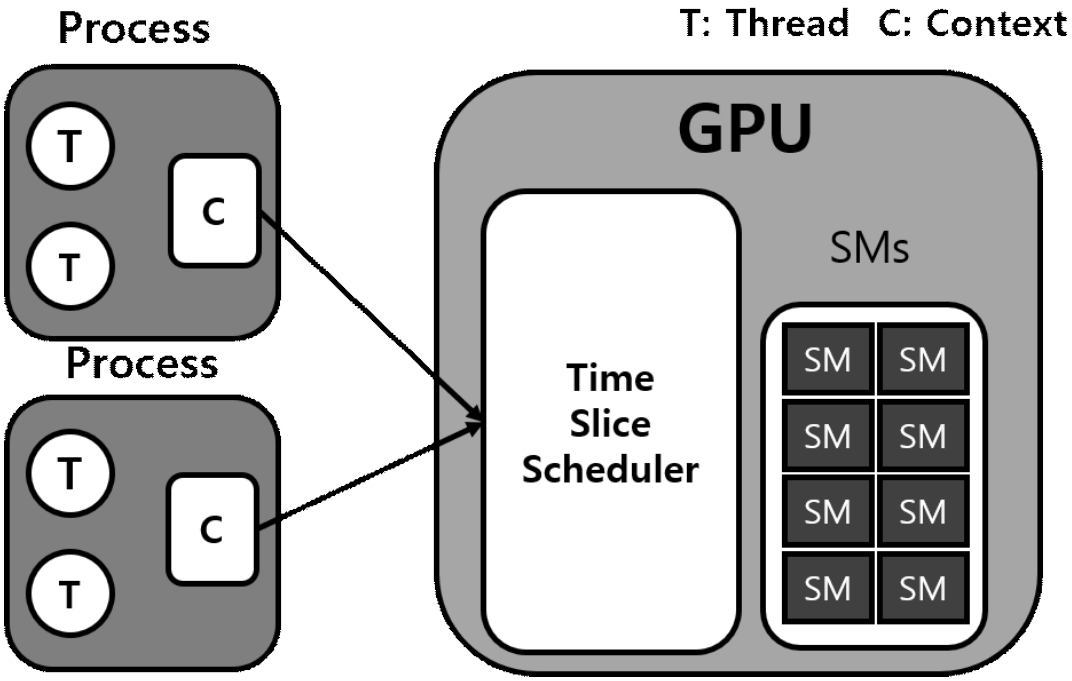
B. & Le, Q. V., 2017)에서는 RNN모델을 컨트롤러로 사용한다. 컨트롤러는 DNN을 생성하고 이 DNN을 CIFAR10, ImageNet과 같은 데이터셋으로 학습시켜 정확도를 평가한다. 이 정확도는 컨트롤러를 학습하는데 사용하며 학습에는 강화학습알고리즘을 사용한다. 컨트롤러는 학습이 진행되면서 점점 정확도가 높은 DNN을 생성하게 된다. [그림 2-1]의 (a)는 레이어 기반 NAS의 DNN구조를 나타낸다. 컨트롤러는 학습이 진행되는 동안 (a)의 각 OP(operation)의 연산자를 선택한다. OP의 후보 연산자들은 탐색공간이라는 이름으로 사용자에 의해 직접 정의된다.

[그림 2-1]의 (a)와 같은 구조는 다양한 구조를 탐색할 수 있지만 레이어가 깊어질 수록 탐색공간이 커질 수 있다는 단점이 있다. 그래서 탐색공간의 크기를 줄이기 위해 NASNet(Zhoh, B., et al, 2018)에서는 [그림 2-1]의 (b)와 같은 셀 기반의 DNN구조를 제안한다. NASNet의 컨트롤러는 모든 레이어의 OP를 선택하지 않고 셀 안의 OP를 선택하여 하나의 셀 구조를 생성한다. 그리고 셀을 여러번 반복하여 쌓아서 DNN을 완성한다. 이 DNN은 여러 개의 셀을 가지고 있으며 모든 셀의 구조는 같다. 이러한 구조는 탐색공간을 줄일 수 있었고 확장성에서도 큰 장점을 얻을 수 있었다. NASNet에서는 CIFAR10으로 탐색한 셀 구조를 몇 개 더 쌓아서 ImageNet에서도 높은 정확도를 얻을 수 있었다.

NAS를 적용하여 MobileNet, ShuffleNet과 같은 모바일, 저전력 임베디드 환경에 적합한 DNN을 탐색한 MnasNet(Tan, M., et al, 2019)에서는 정확도뿐만 아니라 추론 실행시간을 이용하여 컨트롤러를 학습한다. MobileNet과 ShuffleNet은 상대적으로 성능이 부족한 환경을 고려하여 MAC(multiply and accumulate) 연산량을 줄이도록 디자인되었다. 하지만 MnasNet에서는 실제로 DNN이 실행되는 환경에서 추론 실행시간을 측정하여 이를 직접 정한 기준 이하로 줄이는 것에 초점을 둔다. 측정된 추론 실행시간과 정확도를 이용하여 보상을 계산하고 계산된 보상을 이용하여 컨트롤러를 학습시킨다. 컨트롤러는 학습이 진행되면서 추론 실행시간이 빠르면서 높은 정확도를 가지는 DNN을 생성한다. 이를 바탕으로 본 논문에서는 타겟 디바이스인 Xavier와 같은 엣지 디바이스에서 빠르고 높은 정확도를 갖는 DNN을 생성하기 위한

NAS를 제안한다.

제 2 절 GPU의 동작 방식

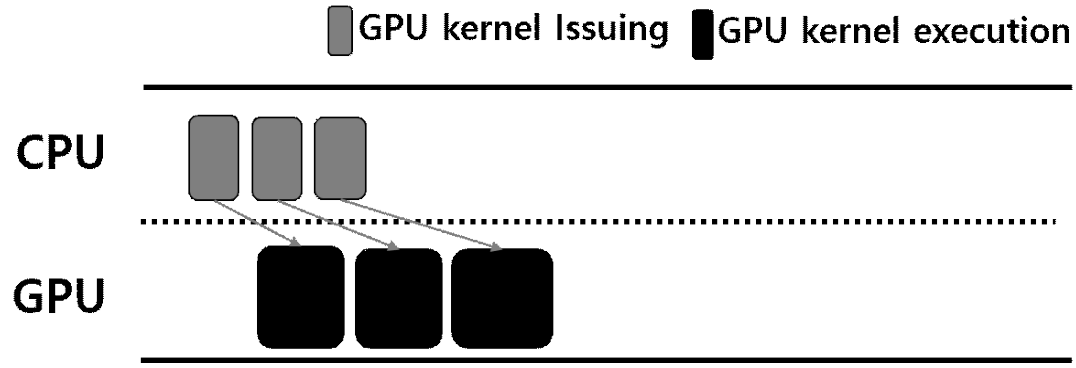


[그림 2-2] GPU를 사용하는 CPU 프로세스의 구조

[그림 2-2]는 GPU를 사용하는 프로세스의 구조를 나타낸다. 일반적으로 CPU는 호스트, GPU는 디바이스라 한다. CPU의 프로세스는 GPU에서 컨텍스트라는 단위로 동작한다. 각 프로세스의 쓰레드에서 전달되는 GPU 작업들은 모두 컨텍스트를 통해서 전달된다. GPU에서는 임의의 한 시점에 하나의 컨텍스트만 활성화되며 일정구간동안 여러 컨텍스트가 존재할 때에는 [그림 2-2]의 Time Slice Scheduler를 통해 컨텍스트에 존재하는 GPU 작업들을 번갈아가며 실행한다.[5)]

[그림 2-3]은 GPU의 함수 단위인 커널함수의 동작을 나타낸다. GPU는 혼자 동작하지 않으며 CPU를 통해 명령을 받아야 함수를 실행할 수 있다.

5) Nvidia multiple-comtexts online:
https://docs.nvidia.com/cuda/cuda-c-best-practies-guide/index.html#multiple-contexts



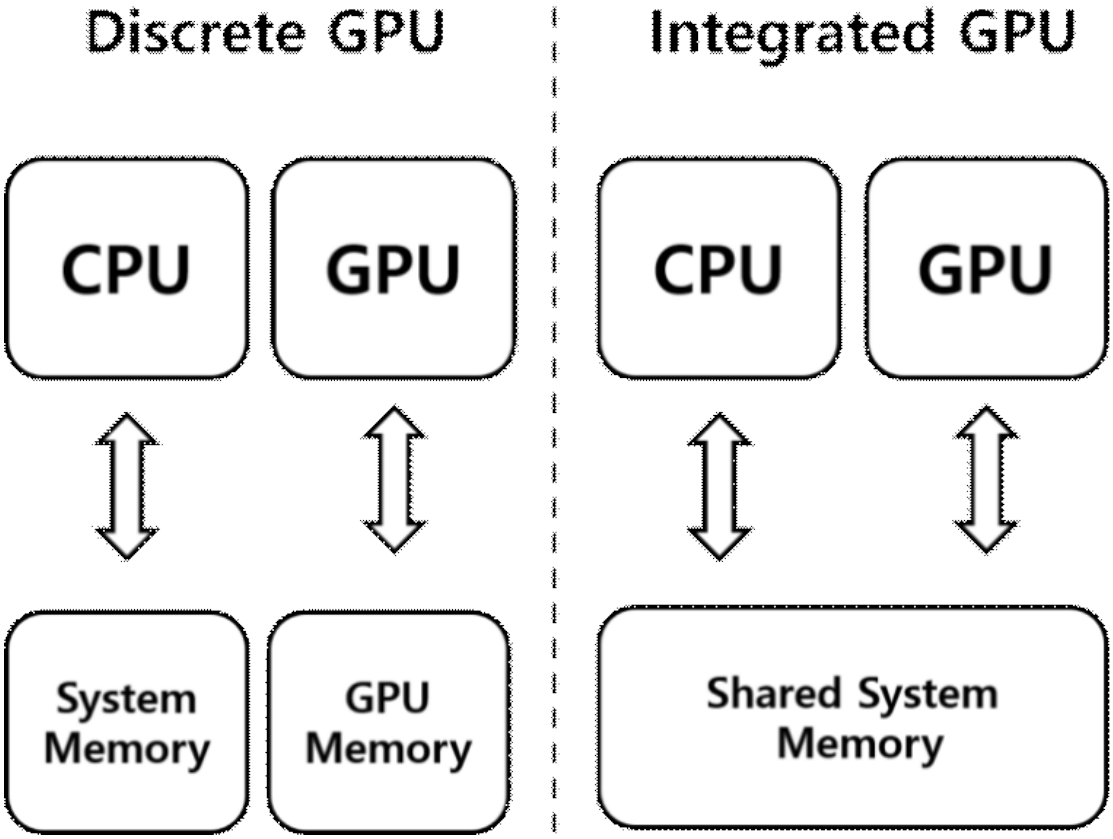
[그림 2-3] GPU 커널함수 동작 방식

커널함수를 실행하려면 다음과 같은 과정을 거쳐야 한다. ① CPU에서 GPU로 실행에 필요한 변수들을 복사한다. ② CPU에서 GPU로 커널함수의 실행정보를 보낸다. ③ CPU에서 받아온 실행정보를 이용하여 GPU에서 커널함수를 실행한다. ④ CPU에서 값을 요청하면 다시 GPU의 변수값들을 CPU 메모리로 복사한다. 이 때 커널함수는 호출된 순서대로 실행되며 CPU와 별도의 실행환경에서 비동기적으로 실행된다. CPU에서 여러 개의 커널함수 실행을 요청하면 GPU에서는 순차적으로 처리하며 커널함수의 종료여부와 관계 없이 CPU에서는 다른 커널함수 실행을 요청하거나 다른 작업을 이어갈 수 있다.

제 3 절 엣지 GPU 디바이스 소개

NVIDIA(사)에서는 오토노머스 머신 및 임베디드 응용프로그램들을 위한 플랫폼인 Jetson 기반의 고급 임베디드 시스템을 제공한다[6]. 본 논문에서는 그 중 임베디드 GPU 기반 시스템 Jetson AGX Xavier(이하 Xavier)를 타겟 디바이스로 사용한다. Xavier는 System-on-Chip(SoC)형태로 설계되어 소형화가 가능하며 소비전력이 적은 특징이 있지만 서버 시스템에 비해 성능이 낮다. [그림 2-4]는 CPU와 GPU의 메모리 구성에 따른 차이를 나타낸다. d-GPU(discrete GPU)형태의 시스템은 CPU와 GPU가 각각 별도의 물리적 메모리 공간 및 캐시를 가지고 있으며 CPU는 시스템 메모리, GPU는 GPU의

6) Nvidia Jetson online:
https://www.nvidia.com/en-us/autonomous-machines/embedded-systems



[그림 2-4] Discrete GPU와 Integrated GPU의 메모리 구성

디바이스 메모리에만 접근이 가능하다. 그에 반해 i-GPU(Intergrated GPU) 형태의 시스템은 CPU와 GPU가 같은 물리적 메모리 공간을 사용하며 각각의 캐시를 가지고 있다. 이 때 물리적 메모리 공간을 CPU와 GPU가 사용량에 따라 나누어 사용한다. 일반적으로 서버시스템은 d-GPU형태로 이루어져 있으며 Xavier는 i-GPU로 이루어져있다. GPU에서 DNN연산을 빠르게 하기 위해 Nvidia에서 제공하는 CUDA Runtime, cuDNN, cuBlas[7], CUDA streams[8] 등의 라이브러리를 활용하여 계산 효율성을 높일 수 있다. Nvidia Jetson Soc 시스템의 엣지 디바이스들은 MPS(Multi-Process Service)[9]와 같

7) Nvidia Cublas, online: https://developer.nvidia.com/cublas

8) Nvidia Cuda Stream, online: https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf

9) Nvidia CUDA Multi-Process Service, online: https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_service_Overview.pdf

	DenseNet-201	Resnet-152	VGG-16	AlexNet
Convolution	201	152	13	5
Fully Conn.	-	-	3	3
Max Pool	4	1	5	3
Avg. Pool	1	1	-	-
SoftMax	1	1	1	1
Dense Conn.	-	98	-	-
Skip Conn.	-	50	-	-
Conv. Ratio	94.3%	95.9%	98.3%	90.8%

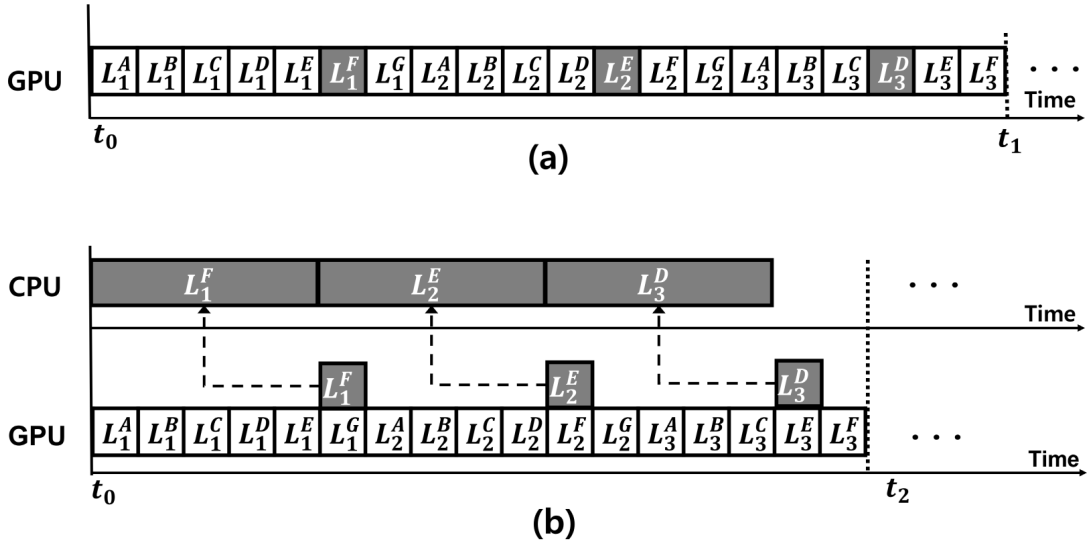
[표 2-1] DenseNet-201, ResNet-152, VGG-16, AlexNet의 레이어 구성

은 멀티 프로세스를 위한 서비스 지원이 부족하다. 따라서 멀티 프로세스를 이용한 GPU 어플리케이션 개발에 사용자가 직접 적절한 스케쥴링 방법을 찾아야 한다.

제 4 절 다중 DNN 실행 분석

1) DNN의 특징 분석

본 논문에서는 다중 DNN 실행을 위해 대표적으로 알려진 4가지의 CNN 기반 DNN인 DenseNet-201(Yu, X., et al, 2019), ResNet-152(He, K., et al, 2016), VGG-16(Simonyan, K. & Zisserman, A., 2014), AlexNet(Krizhevsky, A., et al, 2012)을 사용한다. DNN은 여러개의 레이어로 구성되어 있으며 각 레이어는 연산에 따라 입력값을 특정한 출력값으로 바꾼다. [표 2-1]는 4가지 DNN의 레이어 구성을 보여준다. 표의 각 숫자는 레이어의 숫자를 나타내며 대부분의 레이어는 컨볼루션레이어이다. DenseNet-201의 Dense Conn. 레이어와 ResNet-152의 Skip Conn. 레이어는 다른 일반적인 레이어와 다르게 고유한 연산방식으로 실행된다. 각 모델의 컨볼루션의 연산량을 얻기 위해 Xavier를 사용하여 표에 있는 모든 레이어의 실행시간을 측정했다. 모든 CPU의 성능을 고정하고 한번에 한개의 DNN만 실행하였다. 그리고 스케쥴링에 의한 편차를 제외하기 위해 Linux의 CFS에 (Kim, M., et al, 2018) 구현된 로드밸런싱 알고리즘의 영향을 받지 않도록 측정했다. [표 2-1]의 맨 아래행은 DNN실행 중 컨볼루션 레이어의 실행시



[그림 2-5] 레이어 실행 예시 (a): GPU, (b): CPU-GPU Co-Scheduling

간 비율을 나타낸다.

참고문헌 (Chen, Y.-H., et al, 2017)과 실험에 따르면 DNN연산의 90% 이상은 컨볼루션 레이어가 차지한다. 하나의 컨볼루션 레이어는 단순하게 MAC(multiply-accumulate)연산의 반복으로 구성되기 때문에 병렬 컴퓨팅 기술을 적용하면 더 나은 성능을 기대할 수 있다. 따라서 GPU의 SIMT(single instruction multiple thread)와 참고문헌 (Chen, Y.-H., et al, 2017; Chen, Y., et al, 2014; Higginbotham, S., 2016)과 같은 하드웨어 가속기를 기반으로 하는 텐서연산이 CPU로 대체되어 컨볼루션 레이어의 실행 시간이 크게 단축되었다.

2) CPU와 GPU의 다중 DNN 실행 분석

[그림 2-5]는 DNN(A~G)가 타겟 디바이스에서 실행되는 것을 보여준다. L_1^A은 DNN_A의 첫번째 레이어를 나타낸다. 레이어의 실행 프로세서와 순서는 t_0 이전에 미리 결정된 상태이다. [그림 2-5]의 (a)는 모든 레이어를 GPU에서 실행할 때, (b)는 몇개의 레이어를 CPU에서 실행하고 나머지는 GPU에

서 실행할 때이다. 만약 CPU에서 L_1^F의 실행시간이 GPU에서 L_1^A 부터 L_1^E까지 실행시간의 합과 같거나 작을 때에는 L_1^F를 CPU에서 실행하여 L_1^A과 같은 시간에 실행할 수 있다. 그리고 GPU에서 L_1^G의 시작을 조금 더 앞당길 수 있다. L_2^E, L_3^D에서도 같은 효과가 발생한다. 이렇게 CPU와 GPU를 같이 활용하여 t_1이었던 실행시간을 t_2로 앞당길 수 있다.

DNN의 레이어를 CPU에서도 효율적으로 실행할 수 있지만 CPU와 GPU 간의 메모리를 통한 데이터 전송이 필요하기 때문에 GPU와 CPU의 레이어가 동시에 실행된다고 해도 오버헤드가 여전히 남아있을 수 있다.(Kim, Y., et al, 2019) 따라서 CPU와 GPU를 같이 이용하기 위해서는 두 메모리 공간에 대한 데이터 동기화와 통신 오버헤드 등을 최소화하는 방법이 필요하다.

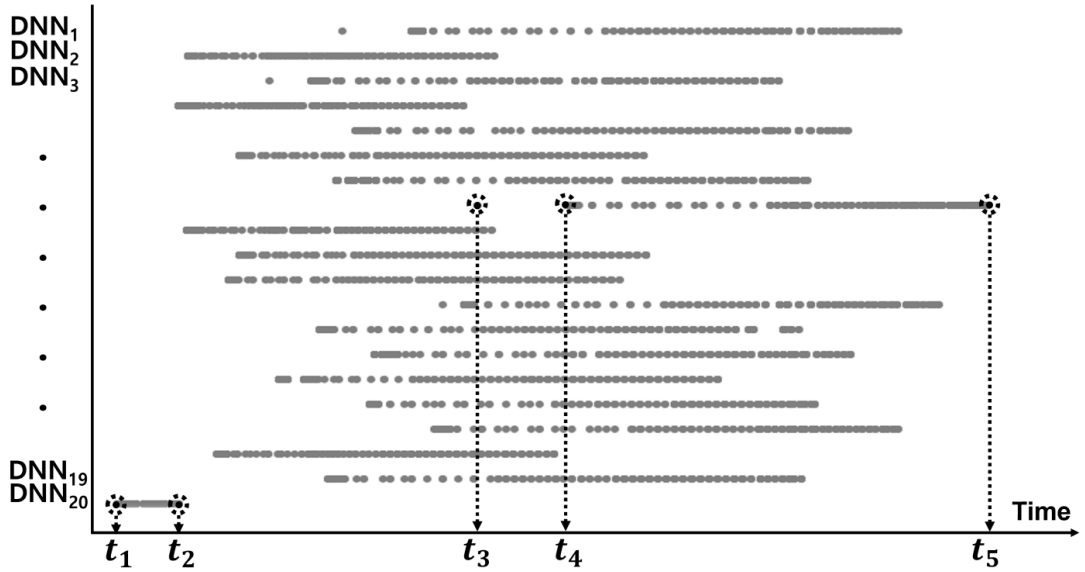
TensorFlow[10], Caffe[11], Torch[12]와 같은 DNN 프레임워크는 DNN모델을 실행할 수 있고 각 모델은 각 프로세스에 포함되어 실행된다. 여러개의 DNN 모델이 동시에 시작될 때, 모델은 같은 시간에 GPU로 실행 명령을 보내며 여러 개의 프로세스가 생성된다. 이 프로세스는 리눅스의 CFS와 같은 스케쥴러에 의해 제어되며 GPU와는 별개로 동작한다. [그림 2-6]은 Xavier에서 DenseNet-201 모델 20개를 GPU에서 동시에 시작했을 때의 결과를 보여준다. 각 모델은 각각의 프로세스에 할당되며 회색 점들은 레이어 연산의 시작 시간을 나타낸다. t_1에서 시작한 DNN 모델은 t_2에서 마지막 레이어를 시작하고 다른 DNN은 t_3에서 첫번째 레이어를 t_5에서 마지막 레이어를 시작한다. 이 때 20개의 DNN이 동시에 시작되었음에도 불구하고 t_5와 t_2간의 차이처럼 종료시간이 크게 차이나는 것을 볼 수 있다. 그리고 프로세스가 다른 프로세스에 의해 선점되면 레이어 함수 발행은 많이 지연될 수 있으며, t_3~t_4 와 같이 유휴시간이 발생하게 된다.

다중 DNN실행을 위해 본 논문에서 해결하려는 문제는 다음과 같다. 1)

10) Tensorflow, online: https://www.tensorflow.org

11) Caffe, online: https://caffe.berkeleyvision.org

12) Torch, onlone: http://torch.ch



[그림 2-6] 동시에 실행된 DNN들의 실행 시작시간 분포

MPS를 지원하지 않는 Xavier의 GPU에서 다중 DNN실행을 위한 연산기법 제시, 2) CPU-GPU간의 원활한 다중 DNN실행을 위해 지연시간을 최소화, 3) CPU-GPU간의 데이터 전송 오버헤드를 최소화 한다. 이 지연시간들을 줄임으로써 DNN모델들의 공정한 스케쥴링을 달성한다.

제 3 장 엣지 디바이스를 위한 NAS 보상함수 제안

본 장에서는 엣지 디바이스에서 작은 추론 실행시간과 높은 정확도를 가진 DNN을 탐색하기 위해 MnasNet의 연구를 기반으로 새로운 다중목적 보상함수를 제안한다.

제 1 절 다중목적 보상함수 제안

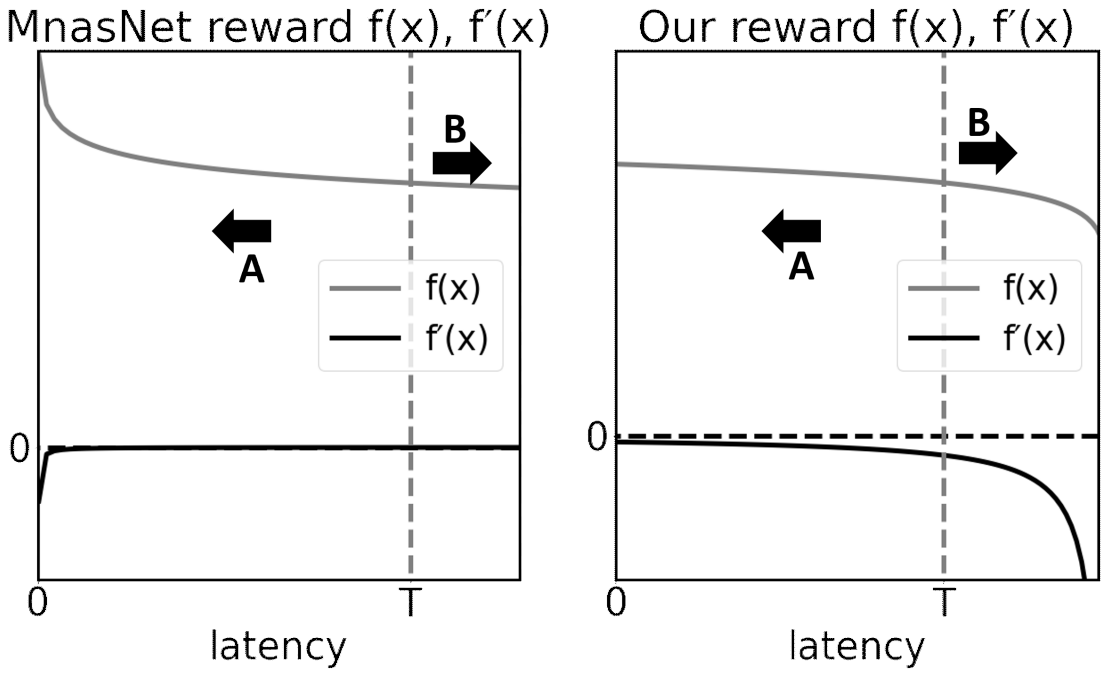
MnasNet(Tan, M., et al, 2019)은 NAS에 다중목적 보상함수를 이용하여 모바일 환경에 맞는DNN구조를 탐색할 수 있었다. 보통 경량화된 DNN들은 MAC 또는 FLOPS를 지표로 사용한다. 하지만 MnasNet에서는 컨트롤러가 생성한 DNN을 직접 타겟 플랫폼에서 실행하여 실행시간을 측정한다. 그리고 실행시간에 대한 기준을 두어, 직접 정한 실행시간보다 낮은 실행시간을 가지는 DNN을 탐색하도록 한다. MnasNet에서 사용하는 다중목적 보상함수는 식 (1)과 같다.

$$\begin{aligned} \underset{m}{\text{maximize}} \quad & ACC(m) \\ \text{subject to} \quad & LAT(m) \leq T \end{aligned} \tag{1}$$

$$\underset{m}{\text{maximize}} \quad ACC(m) \times \left[\frac{LAT(m)}{T} \right]^{w} \tag{2}$$

$$w = \begin{cases} \alpha, & \text{if } LAT(m) \leq T \\ \beta, & \text{otherwise} \end{cases} \tag{3}$$

식 (1)에서 m은 생성된 DNN, ACC는 정확도, 그리고 LAT는 직접 타겟 디바이스에서 측정한 추론 실행시간이다. 이 때 직접 정하는 실행시간의 제한



[그림 3-1] 추론 실행시간에 따른 보상함수 $f(x)$와 미분한 함수 $f'(x)$

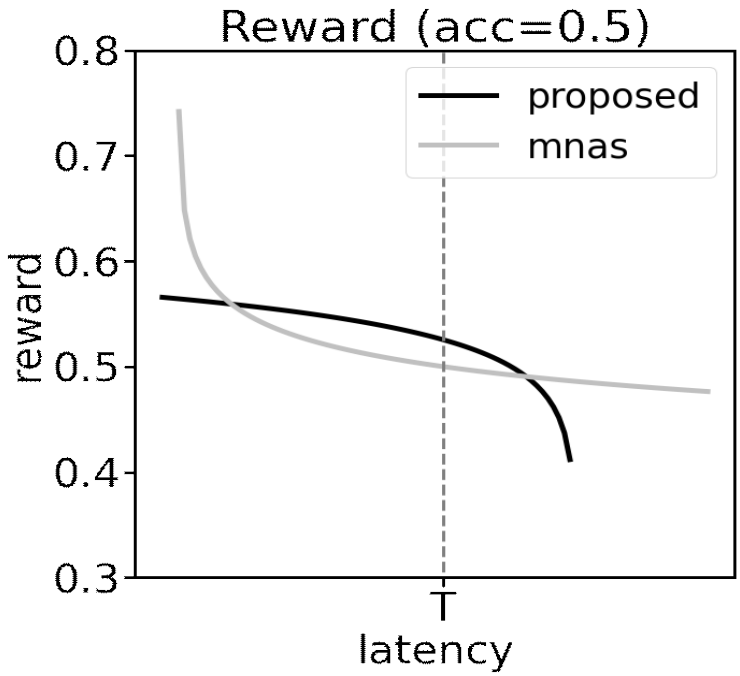
조건은 T이다. 식(1)은 지연시간이 T보다 작으면서 정확도가 최대가 되는 DNN을 찾도록 하는 식이다. MnasNet에서는 이를 위해 DNN의 실행시간과 정확도의 파레토최적을(Deb, K., 2014) 위한 다중목적 보상함수를 식 (2)와 같이 정의한다. 식(2)의 w는 실행시간을 어느정도의 비율로 고려할 것인지에 대한 가중치 값이다. w는 식 (3)에 의하여 정의되며 T를 기준으로 DNN의 추론 실행시간이 클 때와 작을 때 다른 값으로 설정함으로 써 보상의 정도를 조절한다. MnasNet에서는 추론 실행시간과 정확도의 관계를 일정하게 유지하려고 한다. DNN의 실행시간이 두 배 상승 하였을 때 정확도가 5% 상승하는 관계를 유지한다. 이 관계를 유지하기 위해 식 (3)에서 α, β의 값은 모두 -0.07이다. [그림 3-1]의 왼쪽은 MnasNet의 보상함수와 이를 미분한 함수의 그래프이다. x축은 실행시간이며 y축은 보상값이고 정확도는 50%로 고정하여 실행시간에 따른 보상값의 변화를 보여준다. T의 값은 17ms이며 이후 언급이 없는 한 T는 17ms로 고정한다. MnasNet의 보상함수는 DNN의 추론 실행시간이 줄어드는 방향(A)로 갈 수록 보상의 증가량이 커진다. 반대로 추론 실행시간이 늘어날 수록 보상의 감소량이 줄어든다. 따라서 같은 정

확도를 가지는 두 DNN의 추론 실행시간이 15ms, 5ms 일 때가 20ms, 10ms 일 때 보다 보상값의 차이가 더 크다. 실행시간이 T를 넘는 DNN에 대한 보상의 감소량이 작기 때문에 MnasNet에서는 추론 실행시간이 T를 넘는 DNN도 생성을 많이 한다.

$$latency(m) = \begin{cases} \log_T(-LAT(m) + 1.5T), \\ \quad \text{if } LAT(m) \leq 1.5T - 1 \\ 0, \ otherwise \end{cases} \tag{4}$$

$$\underset{m}{\text{maximize}} \quad x \times ACC(m) + y \times latency(m) \tag{5}$$

우리는 MnasNet과 같이 추론 실행시간을 이용한 새로운 다중목적 함수를 제안한다. 제안하는 다중목적 함수는 정해진 T보다 낮은 추론 실행시간을 가지면서 정확도가 가장 높은 DNN을 더 빠르게 탐색할 수 있도록 설계되었다. 식 (4)와 식 (5)는 제안하는 보상함수를 나타낸다. 식 (4)는 추론 실행시간에 대한 식이다. 식 (5)는 정확도와 추론 실행시간에 가중치 합 형태의 파레토 최적 보상함수이다. 식 (5)에서 x는 정확도, y는 실행시간의 비율을 조절할 수 있는 변수이며 x+y=1의 조건에 맞게 설정한다.(Deb, K., 2014) 본 논문에서는 x=0.9, y=0.1로 설정하였다. [그림 3-1]의 오른쪽은 제안하는 보상함수에 대한 그래프이다. 추론 실행시간이 줄어드는 방향(A)으로 이동할 때 보상의 증가량이 줄어들고, 추론 실행시간이 늘어나는 방향(B)로 이동할 때 보상의 감소량이 늘어난다. MnasNet과 반대로 같은 정확도의 두 DNN의 추론 실행시간이 15ms, 5ms일 때 보다 20ms, 10ms일 때의 보상의 차이가 더 크다. DNN의 추론 실행시간이 T보다 작다면 보상의 증가량을 제한함으로써 추론 실행시간에 대한 보상의 정도를 줄이는 것이다. T는 직접 설정하는 값이고 예를들어 30fps를 만족하기 위해서는 한 프레임당 처리 속도가 33ms만 넘지 않으면 된다. 따라서 추론 실행시간이 제한 조건인 33ms보다 작다면 정확도가 높은 DNN이 가장 좋다고 말할 수 있다. 만약 정확도와 추론 실행시간이 DNN1은 60%, 8ms이고 DNN2는 63%, 16ms라고 하자. 이 때



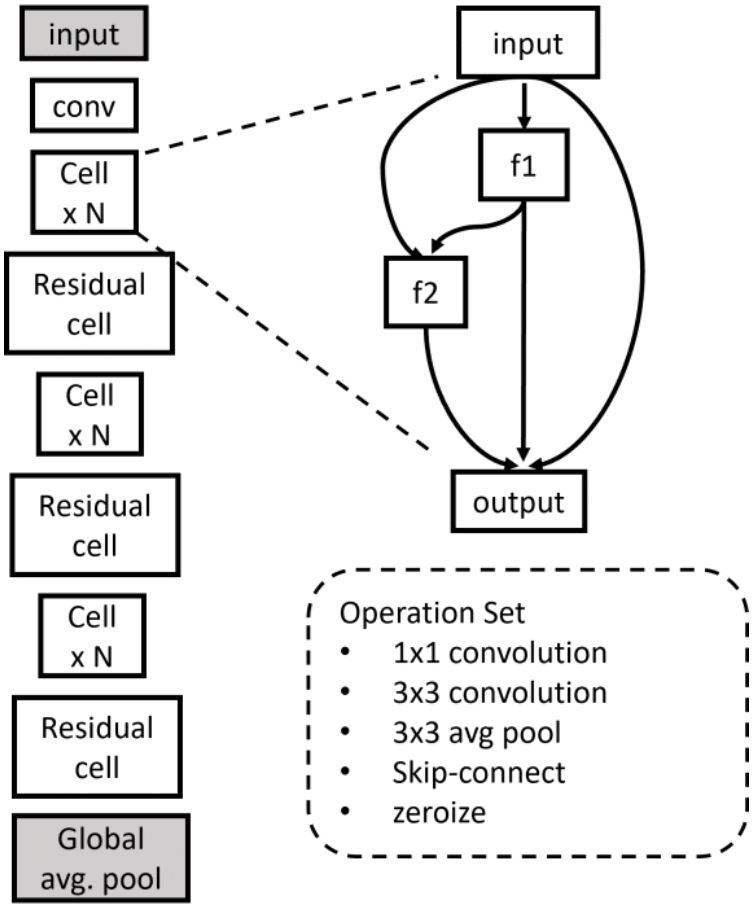
[그림 3-2] 보상 비교

MnasNet의 보상함수로 계산한 결과값은 DNN1과 DNN2가 비슷하지만 본 논문에서 제안하는 보상함수는 두 DNN이 T 보다 작은 추론 실행시간을 가지므로 정확도가 더 큰 DNN2에 더 높은 보상을 준다.

[그림 3-2]에서는 추론 실행시간에 따른 두 보상함수를 비교한 그림이다. 추론 실행시간이 T보다 클 경우에는 보상의 감소량을 증가시켜 T보다 높은 DNN을 탐색하지 않도록 유도한다. 그리고 추론 실행시간이 T보다 작을 때에는 MnasNet과 다르게 보상을 완만하게 증가시켜 너무 작은 추론 실행시간에 큰 보상을 주지 않도록 조절한다.

제 2 절 NAS-Bench-201 및 NAS 제안

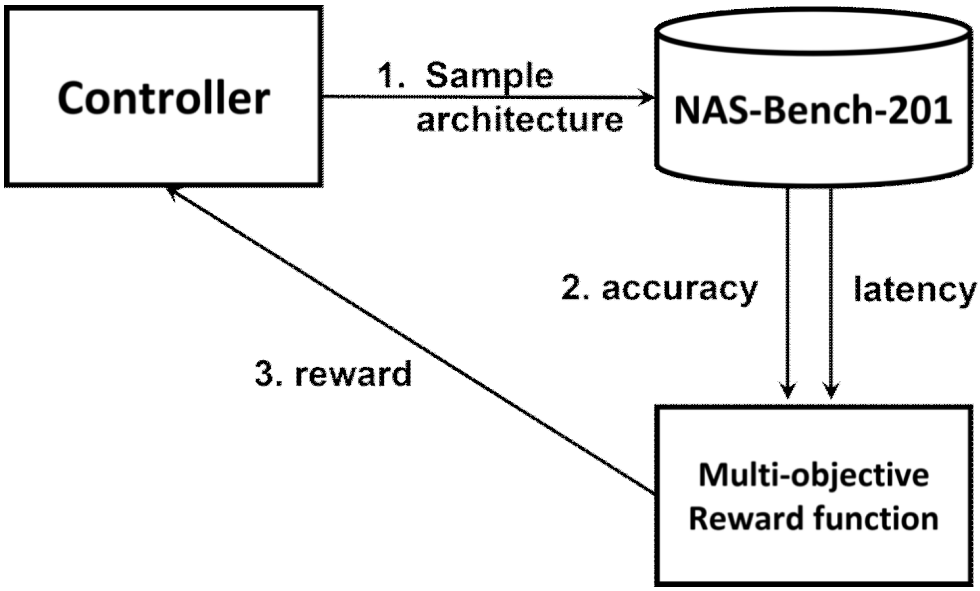
NAS에서 가장 큰 제한 사항이 DNN을 탐색하는 시간이 상당히 오래 걸린다는 것이다. 이는 생성되는 DNN들의 정확도를 평가하기 위해 CIFAR-10이나 ImageNet 데이터셋에 학습시켜야 하기 때문이다. 이는 데이터셋의 크기가 클 수록 기하급수적으로 증가한다. 처음 등장한 NAS는 800개



[그림 3-3] NAS-Bench-201의 DNN 구조

GPU를 이용해도 28일의 시간이 소요되었고(Zops, B. & Le, Q. V., 2017), 셀 기반으로 탐색공간을 줄인 NasNet에서도 500개의 GPU로 4일동안 탐색하였다.(Zhoh, B., et al, 2018) 위 두 연구에서는 CIFAR-10을 이용하였지만 MnasNet에서는 ImageNet으로 학습시키기 위해 5epoch만 학습시켰음에도 불구하고 TPU 64개를 이용하여 45일 동안 탐색하였다(Tan, M., et al, 2019).

NAS-Bench-201(Dong, X. & Yang, Y., 2020)은 NAS를 위한 DNN의 정확도와, 학습시간, 구조 등을 모아놓은 벤치마크용 데이터셋이다. 이 결과들은 미리 측정하여 저장해놓은 값이므로 따로 학습할 필요 없이 컨트롤러가 생성하는 DNN의 구조만 넣으면 바로 정확도를 얻을 수 있다. 따라서 NAS-Bench-201을 이용하면 NAS에서 가장 큰 시간을 차지하는 부분을 줄



[그림 3-4] 제안된 다중목적 보상함수가 적용된 NAS

일 수 있다. [그림 3-3]은 셀 구조 기반으로 이루어진 NAS-Bench-201의 DNN구조를 나타낸다. 각 셀은 [그림 3-3]의 우측 과 같이 정의된다. 각 화살표는 하나의 연산(Operator)를 나타내며 이 연산의 후보군은 Operation Set에 정의되어있다. f1, f2는 각각 중간 피쳐맵을 나타내고 화살표의 끝 부분이 모이는 곳에서는 각 피쳐맵에 대한 concat연산으로 하나로 합친다. Cell과 Residual Cell 둘 다 같은 셀 구조를 사용하며 Cell의 stride는 1, Residual Cell의 stride는 2로 설정하여 피쳐맵의 크기를 조절한다. 각 단계에서 Cell을 N번 반복하며 이는 5로 설정되어있다. NAS-Bench-201의 후보 DNN 수는 총 15,625가지이며 각 DNN에 대한 정확도, 실행시간, 용량, FLOPS(floting point operations per second)를 이미지 데이터셋 (CIFAR-10, CIFAR-100 (Krizhevsky, A. & Hinton, G., 2009), ImageNet-16-120(Chrabaszcz, P., et al, 2017))에서 평가한 결과를 포함하고 있다.

본 연구에서 제안하는 다중목적 보상함수 기반의 NAS를 [그림 3-4]과 같이 나타낸다. 각각 컨트롤러, NAS-Bench-201, 보상함수로 이루어져 있으며 컨트롤러는 RNN으로 구성했다. 컨트롤러는 NAS-Bench-201의 DNN구조에 맞게 DNN을 생성하고, 생성된 DNN을 NAS-Bench-201의 입력으로 사용하여 정확도와 추론 실행시간을 얻는다. 이 정확도와 추론 실행시간을 보상함

수의 입력으로 넣어 보상값을 계산하고 이 보상값으로 컨트롤러를 학습한다. 컨트롤러의 학습은 강화학습 알고리즘 중 Policy Gradient(PG)(Williams, R. J., 1992)를 이용한다.

제 4 장 다중 DNN 추론 최적화 기법 제안

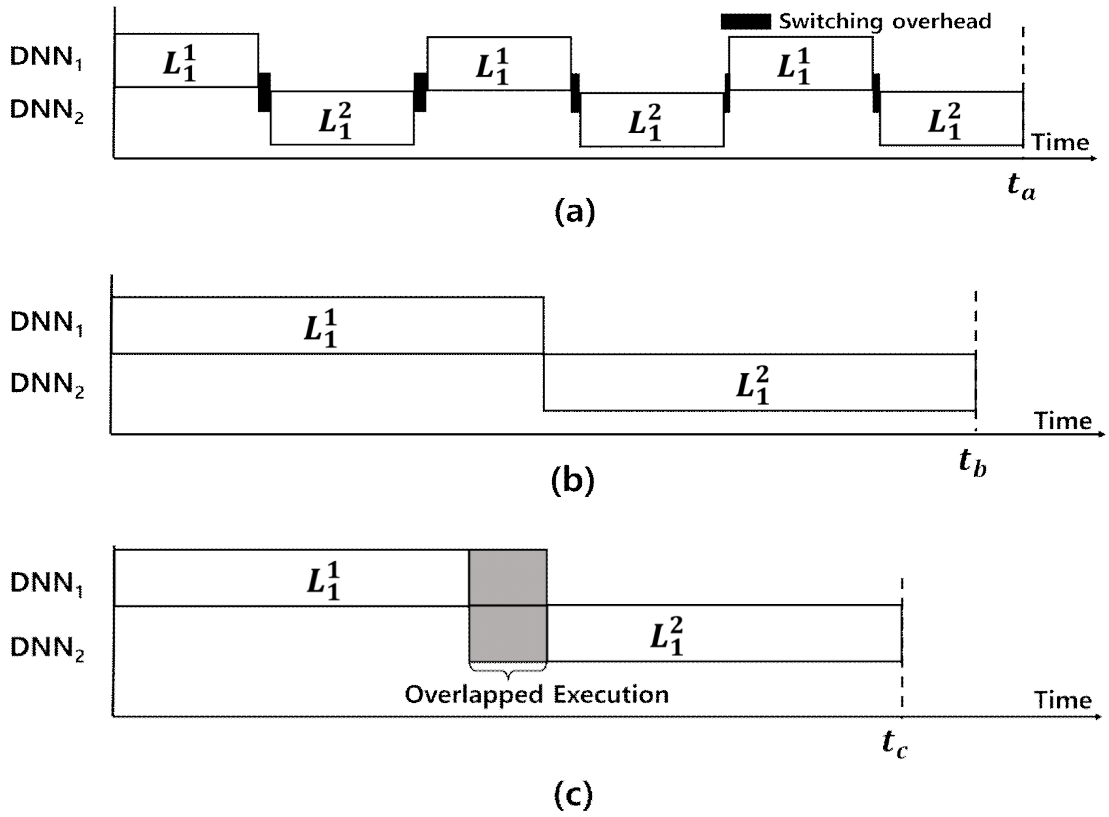
본 장에서는 SOTA(State Of The Art)를 만족하고 널리 쓰이는 DNN모델들을 이용하여 여러 개의 DNN을 동시에 실행할 때 빠르고 실행시간의 편차를 줄일 수 있는 방법을 제안한다. 각 DNN은 연속된 레이어들의 집합이며 각 레이어는 GPU커널함수로 구현된다. n번째 DNN인 DNN_n가 k개의 레이어로 이루어져 있을 때 $DNN_n = \left\{L_1^n, L_2^n, \cdots, L_k^n\right\}$ 와 같이 표현한다. 여기서 L의 윗첨자는 DNN의 id, 아랫첨자는 레이어의 id를 나타낸다.

제 1 절 엣지 디바이스 GPU 오버헤드 최적화 기법 제안

1) 멀티 DNN 추론 환경 및 오버헤드

2장에서 언급한 것과 같이 프로세스는 CUDA에서 컨텍스트라 불리며 모든 동작은 컨텍스트 단위로 동작한다. 프로세스에 여러 쓰레드가 존재하면 이 쓰레드들은 하나의 컨텍스트에서 동작한다. 그리고 프로세스에서 여러개의 컨텍스트가 동작하도록 설정할 수 있지만 CUDA에서는 프로세스당 하나의 컨텍스트를 권장한다. 임의의 한 시점에는 GPU에 하나의 컨텍스트만 활성화되기 때문에 여러개의 컨텍스트가 활성화 되어있는 상태에서는 Time Slice Scheduler가 각 컨텍스트를 스케쥴링하여 GPU를 점유한다. 이때 컨텍스트를 번갈아가면서 실행하기 때문에 기존 컨텍스트의 정보들을 메모리에 저장하고 다음 컨텍스트의 정보를 GPU에 할당하는 컨텍스트 교환을 하게 되는데 이때의 오버헤드를 컨텍스트 교환 오버헤드라고 한다. 서버시스템에서는 멀티 컨텍스트 환경에서 컨텍스트 교환 오버헤드를 줄일 수 있도록 앞에 언급한 MPS를 제공해 주지만 우리의 타겟 디바이스인 Xavier에서는 MPS를 지원하지 않는다.

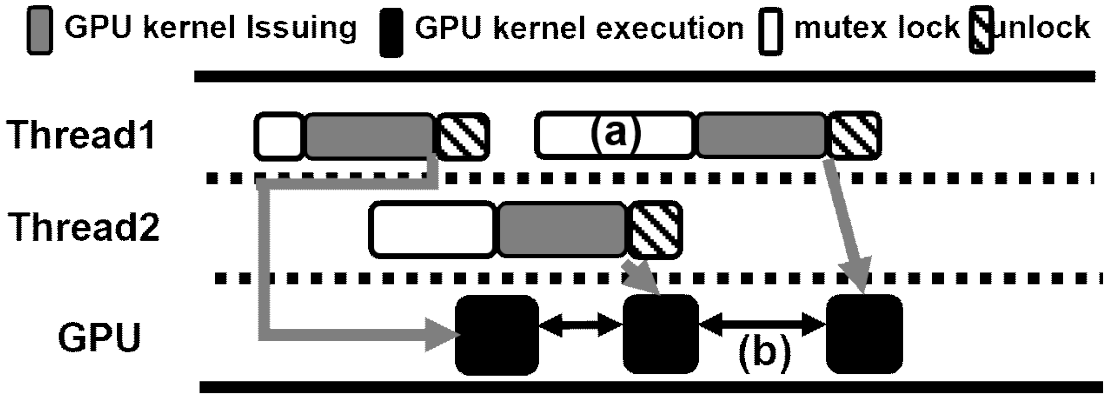
컨텍스트 교환 오버헤드는 GPU에 여러 개의 컨텍스트가 존재할 때 발생한다. 컨텍스트가 여러개 존재하면 멀티 컨텍스트 환경, 하나만 존재하면 싱



[그림 4-1] GPU스케쥴링 방식에 따른 레이어 실행시간 비교

글 컨텍스트 환경이라 한다. [그림 4-1]의 (a)와 (b)는 각각 멀티 컨텍스트 환경과 싱글 컨텍스트 환경에서 DNN_1 과 DNN_2 가 동작하는 것을 보인다. 두 DNN의 레이어 L_1^1, L_1^2 가 각 환경에서 동작할 때, 멀티컨텍스트 환경인 (a)에서는 레이어가 실행되는 동안 일정 시간마다 컨텍스트 교환이 일어난다. 컨텍스트 교환이 일어날 때마다 일정한 오버헤드가 있어서 실행시간이 지연된다. 반면에 싱글 컨텍스트 환경인 (b)에서는 하나의 레이어 실행이 끝난 후 다음 레이어를 실행하기 때문에 컨텍스트 교환 오버헤드가 없다. 따라서 t_a보다 t_b가 더 빠르게 끝나는 것을 확인할 수 있다. (c)는 CUDA streams을 이용했을때 얻을 수 있는 병렬 효과를 보여준다. CUDA streams을 이용하면 GPU 자원이 남는 한 다른 CUDA streams에 존재하는 여러개의 커널함수를 실행할 수 있기 때문에 병렬 처리 효과를 얻을 수 있다.

싱글 컨텍스트 환경에서 다중 DNN을 실행하기 위해 각 DNN을 쓰레드

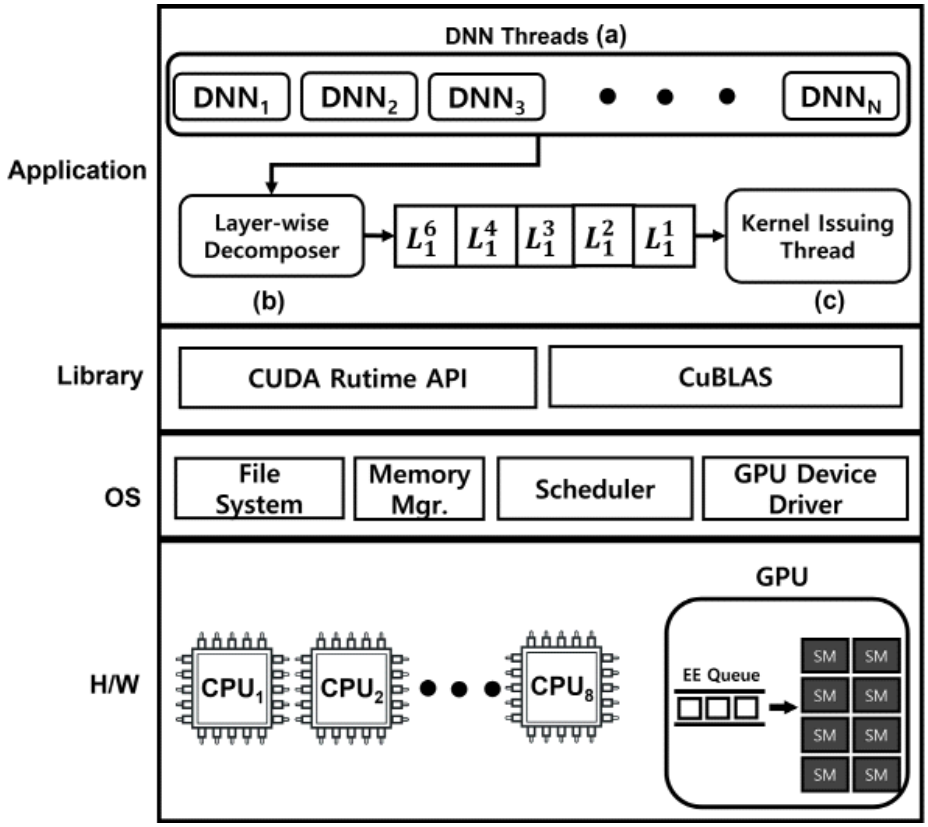


[그림 4-2] 싱글 컨텍스트 환경의 mutex 오버헤드

로 실행한다고 가정한다. 그러면 각 쓰레드들은 하나의 컨텍스트를 통해 GPU로 커널함수를 발행하게 된다. [그림 4-2]는 각 쓰레드가 하나의 컨텍스트로 커널함수를 발행할 때의 동작을 보여준다. 여러 쓰레드가 GPU에 커널함수를 발행하기 위해 접근하는데, 이 때 CUDA API는 GPU를 공유자원으로써 동기화 함수인 mutex를 이용하여 여러 쓰레드가 동시에 접근할 수 없도록 제한한다. 여러 개의 쓰레드가 공유자원 GPU에 커널함수를 발행하기 위해 동시에 접근을 시도하면 각 쓰레드들은 mutex 획득/해제를 수행하여 직렬화(serialize)구간이 늘어난다. CPU에서 실행되는 함수가 mutex 획득/해제를 하는 동안에는 GPU에 아무런 작업이 없어도 커널함수를 발행할 수 없기 때문에 유휴시간이 발생되어 CPU의 작업 때문에 GPU를 원활하게 사용할 수 없는 문제가 발생한다. [그림 4-2]는 mutex 획득/해제 오버헤드가 주는 영향을 보여준다. mutex로 인해 (a) 오버헤드로 커널함수의 발행이 지연되면 (b)와 같이 GPU에 유휴시간이 생긴다. 이 오버헤드를 싱글 컨텍스트 오버헤드라 한다.

2) 멀티 DNN 추론을 위한 GPU 오버헤드 최소화 프레임워크 제안

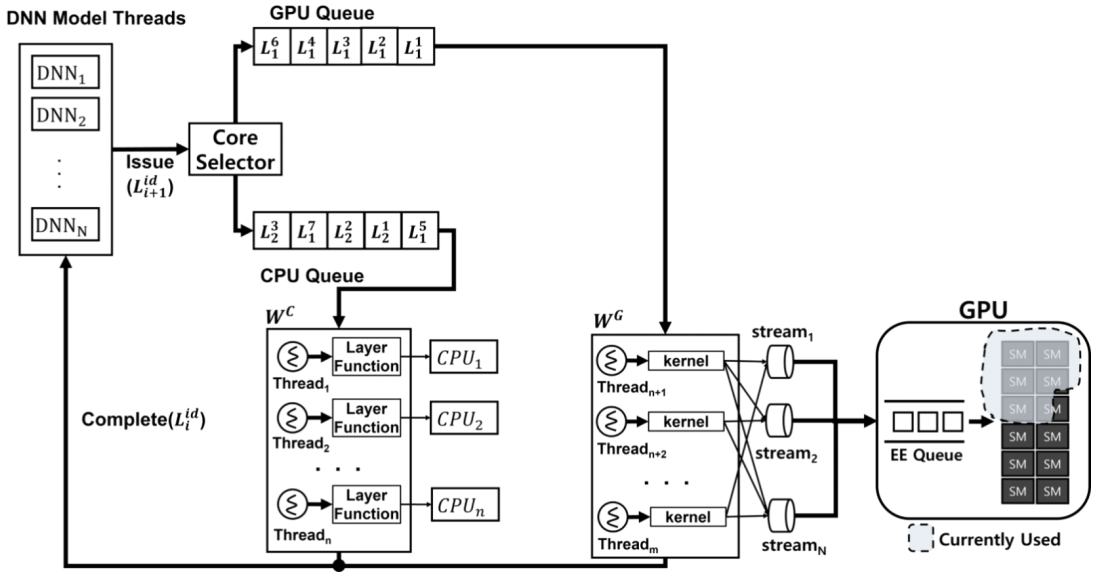
여기서는 앞에 언급한 두 가지 오버헤드를 최소화 하는 프레임워크를 제안한다. [그림 4-3]는 앞에서 언급한 두가지 오버헤드인 멀티 컨텍스트 오버헤드, 싱글 컨텍스트 오버헤드를 최소화 하기 위한 프레임워크의 구조이다. 첫 번째로 멀티 컨텍스트 오버헤드를 줄이기 위해 하나의 컨텍스트만 사



[그림 4-3] 오버헤드가 최소화된 다중 DNN 실행을 위한 구조

용한 싱글 컨텍스트 환경을 사용한다. 하나의 프로세스에 각 DNN쓰레드는 (a)와 같이 배치되며 싱글 컨텍스트 오버헤드를 피하기 위해 GPU에 접근하는 커널발행 담당쓰레드(c)를 하나만 배치한다. 각 DNN은 GPU 커널함수를 발행하기 위해서는 쓰레드(c)에 접근해야 하며 여러 레이어를 동시에 처리할 수 없기 때문에 큐를 이용한다. 각 DNN 쓰레드는 큐로 레이어 실행정보를 보내 대기시키고 쓰레드(c)는 하나씩 꺼내어 순차적으로 처리한다. 이러면 GPU에는 쓰레드(c)만 접근하게 되어 싱글 컨텍스트 오버헤드가 사라진다.

본 프레임워크는 다음과 같이 작동한다. (a)의 DNN 쓰레드들은 레이어를 Layer-wise Decomposer(b)로 보낸다. (b)는 상대적으로 실행시간이 작은 Batch Normalization, Activation 등의 레이어들을 Convolution 레이어와 하나로 묶어 큐로 전달한다. 큐도 공유자원이므로 큐에 접근하는 횟수가 많아질수록 큐에 대한 mutex인 Qmutex로 인한 오버헤드가 늘어난다. 따라서 (b)로 레이어를 모아서 전달하면 큐에 접근하는 횟수가 줄어들어 Qmutex 획득/해



[그림 4-4] 다중 DNN 실행을 위해 CPU-GPU Co-Scheduling을 적용한 구조

제로 인한 오버헤드가 줄어들게 된다. 그리고 (c)는 큐의 레이어를 하나씩 GPU 커널함수로 발행한다. 커널함수는 GPU의 스트림에서 대기하다가 스트림 가장 앞의 함수를 EE Queue에 대기시킨다. GPU는 EE Queue의 커널함수들을 순차적으로 처리한다.

제 2 절 CPU-GPU Co-Scheduling 프레임워크 제안

1) CPU-GPU Co-Scheduling 프레임워크

우리는 다중 DNN 실행을 위한 CPU-GPU Co-Scheduling 프레임워크를 제안한다. 1절에서 언급한 오버헤드를 줄이기 위해 하나의 프로세스의 멀티 쓰레드로 구성된다. [그림 4-4]는 제안하는 프레임워크의 워크플로우를 보여준다.

프레임워크는 DNN 모델 쓰레드집합, 코어 선택기, CPU실행을 위한 워커 쓰레드의 집합인 쓰레드풀 W_C와 대기열 CPU-큐, GPU실행을 위한 워커 쓰레드의 집합인 쓰레드풀 W_G와 대기열 GPU-큐로 구성된다. DNN 모델 쓰레드 집합은 N개의 DNN이 존재하며 $\{DNN_1, DNN_2, \cdots, DNN_N\}$과

같이 표현된다. DNN의 각 숫자는 DNN 구별을 위한 id값이며 N개의 DNN이 있으면 id값은 $1 \leq id \leq N$ 이다. DNN_{id}의 i번째 레이어 작업을 시작할 준비가 끝나면 코어 선택기에 $Issue(L_i^{id})$신호를 보낸다. 여기서 레이어 작업은 레이어의 가중치와 같은 실행정보를 포함하는 일종의 데이터 구조를 의미한다. 코어 선택기는 레이어 작업을 실행하기 적합한 코어 유형을 선택한다. 이 때 코어선택을 위한 짧은 지연시간을 가진 알고리즘이 동작하며 이 알고리즘은 다음 항에서 설명한다. 선택이 끝나면 레이어 작업은 CPU-큐 또는 GPU-큐중 선택된 코어의 큐에서 대기한다. 각 큐의 레이어 작업의 가장 앞에 있는 레이어 작업은 각 코어의 워커 쓰레드로 보내진다. 두 코어의 쓰레드풀 W_C와 W_G는 각각 n개와 m개의 쓰레드를 포함한다. 모든 쓰레드(n+m)는 CPU의 코어에 고정되어 CPU간 이동에 의해 유발되는 예상하지 못한 오버헤드의 영향을 받지 않도록 한다. W_C에 포함된 워커 쓰레드중 작업중이 아닌 워커 쓰레드가 있을 때 W_C는 CPU-큐의 가장 앞에 존재하는 레이어 작업을 작업중이 아닌 쓰레드에 할당한다. 이 할당 방식은 W_G의 워커 쓰레드에서도 동일하게 적용된다.

하지만 W_C와 W_G의 레이어 작업 실행은 다른 방법으로 실행된다. W_C의 워커 쓰레드는 일반적인 CPU실행과 같이 전달된 레이어 작업을 실행한다. 반면에 W_G의 워커 쓰레드는 레이어 작업에서 GPU 커널함수를 사용가능한 CUDA Stream에 매핑한다. CUDA Stream을 통해 GPU로 전달된 레이어 작업은 여러 SM들을 동시에 실행한다. GPU로 전달된 커널함수는 GPU의 EE Queue(execution engine queue)에서 대기하며 이는 GPU-큐와 다르게 오직 GPU 커널함수 실행을 위한 큐이다.

마지막으로 W_C의 워커 쓰레드중 하나의 레이어 작업 실행이 끝나면 DNN모델 쓰레드에 $Complete(L_i^{id})$ 신호를 전달하고, 그 다음 레이어 작업에 대한 $Issue(L_{i+1}^{id})$신호가 코어 선택기로 전달된다. 그런데 W_G의 경우엔 다르게 동작한다. W_G의 워커 쓰레드는 CPU와 비동기적으로 동작하는 GPU

의 특성 때문에 L_i^{id}를 CUDA Stream에 매핑한 다음 레이어 작업의 종료 여부에 관계 없이 바로 $Complete(L_i^{id})$ 신호를 전달한다. 이 경우 GPU에서 L_i^{id}를 실행하는 동안 다음 작업을 대기시킬 수 있다. 이러한 $Complete(L_i^{id})$ 신호를 통해 DNN의 레이어 실행 순서를 유지할 수 있다.

2) 코어 선택 알고리즘

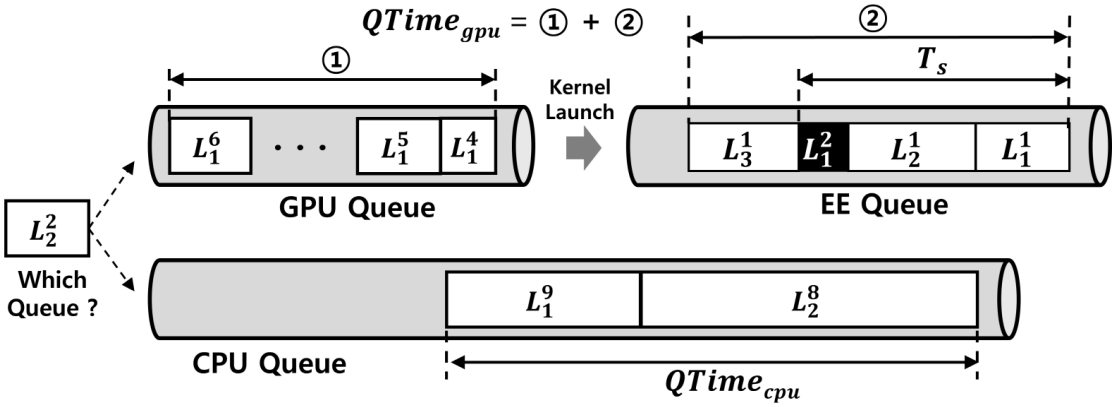
다중 DNN실행을 위한 프레임워크를 시작하면 CPU-Q와 GPU-Q의 대기열을 비우고 W_C와 W_G의 모든 쓰레드들이 각 큐의 가장 앞에 있는 레이어 작업을 가져올 준비를 한다. [그림 4-5]은 코어 선택기의 동작과 CPU/GPU 큐 관리 정책에 대한 의사 코드를 설명한다. 알고리즘의 CoreSel 함수는 코어 선택기의 동작을 설명한다. 코어 선택기의 목표는 해당하는 레이어 작업 L_i^{id}에 가장 적합한 코어를 찾아 해당하는 큐에 푸시하는 것이다. 레이어 작업 L_i^{id}이 코어 선택기로 전달되면 레이어 작업의 (i, id)에 따라 CPU와 GPU에서의 실행시간을 예측한다. 실행시간을 예측하기 위한 예측 모델에 대한 내용은 이후에 설명한다.

알고리즘의 라인3-4에 정의된 것과 같이 각 코어에는 고유한 예측 모델이 있으며, 예측된 실행시간은 T_C와 T_G로 표시한다. 그리고 앞에서 언급한 것과 같이 GPU에 레이어 작업을 위한 커널함수를 발행한 후에는 바로 다음 작업을 실행할 수 있다. 예를 들어 L_1^2가 GPU에서 보류중인 경우에도 CPU는 다음 레이어 작업인 L_2^2에 대해 $Issue$를 할 수 있다. [그림 4-6]는 이 동작을 설명하기 위한 그림이다. [그림 4-6]에서 L_i^{id}의 하얀 상자들의 폭은 예측 모델을 통해 얻은 예측된 실행시간을 나타낸다. 만약 L_2^2가 CPU-큐에 연결되면 GPU에 있는 L_1^2의 실행이 완료될 때 까지 L_2^2을 실행할 수 없기 때문에 T_C에 T_S가 추가된다.([그림 4-5], 라인 5~7) T_C와 T_G가 결정되면 레이어 작업은 더 짧은 대기시간을 가지는 큐에서 대기하며 코어 선택 결과

Algorithm 1 Core Type Selection and Queue Management

Function CoreSel(L_i^{id})
$T_c \leftarrow 0, T_g \leftarrow 0, T_s \leftarrow 0$
$T_c \leftarrow$ **LR_Model**$_{cpu}(i, id)$
$T_g \leftarrow$ **LR_Model**$_{gpu}(i, id)$
if core_type of the L_{i-1}^{id} == GPU **then**
    $T_s \leftarrow$ Time for Synch. if L_i^{id} is on the CPU
    $T_c \leftarrow T_c + T_s$
end if
if $QTime_{gpu} + T_g \geq QTime_{cpu} + T_c$ **then**
    $T_i^{id} \leftarrow T_c$, and $QTime_{cpu} \leftarrow QTime_{cpu} + T_c$
    push L_i^{id} to the CPU queue
else
    $T_i^{id} \leftarrow T_g$, and $QTime_{gpu} \leftarrow QTime_{gpu} + T_g$
    push L_i^{id} to the GPU queue
    attach E_i^{id} to L_i^{id}
end if
Function LayerDeQ(core_type)
if core_type == GPU **then**
    $L_i^{id} \leftarrow$ popped from the GPU queue
else if core_type == CPU **then**
    $L_i^{id} \leftarrow$ popped from the CPU queue
    $QTime_{cpu} \leftarrow QTime_{cpu} - T_i^{id}$
end if
return L_i^{id}
Function EventCheckGPU( )
while E_i^{id} is not completed **do**
    do nothing
end while
$QTime_{gpu} \leftarrow QTime_{gpu} - T_i^{id}$

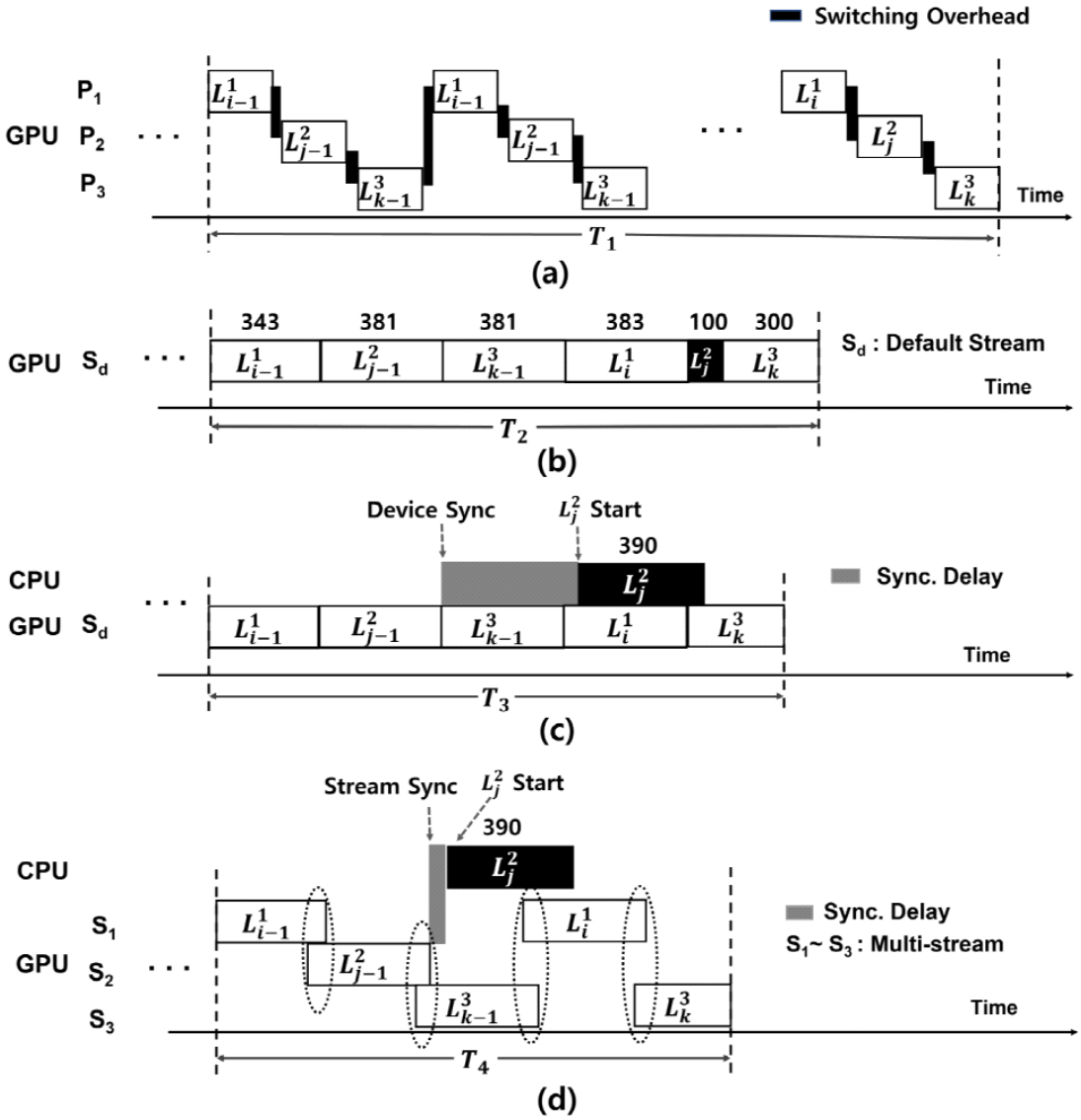
[그림 4-5] CPU-GPU Co-Scheduling을 위한 알고리즘



[그림 4-6] 새로운 레이어 작업에 대한 코어 선택기 예시

에 따라([그림 4-5], 라인 9~16) $QTime_{cpu}$ 또는 $QTime_{gpu}$의 값을 업데이트한다. 여기서 $QTime_{cpu}$와 $QTime_{gpu}$는 각각 CPU-큐, GPU-큐의 총 예상 대기시간이다. 만약 L_2^2가 GPU-큐에 연결되면 $QTime_{gpu}$는 이미 T_S가 포함되어 있기 때문에 T_G에 T_S를 추가하지 않는다. L_i^{id}가 큐에 푸시할때마다 대응되는 예상 실행시간 T_i^{id}는 메인 메모리에 저장된다.

W_C의 워커 쓰레드가 CPU-큐에서 L_i^{id}을 꺼내면 $QTime_{cpu}$에서 T_i^{id}를 차감하여 업데이트한다.([그림 4-5], 라인 20~22) 반면에 L_i^{id}가 GPU에서 실행될 때에는 L_i^{id} 바로 실행되지 않기 때문에 $QTime_{gpu}$는 변하지 않는다. 대신 GPU SM내부 작업을 위한 EE-Queue로 푸시된다. 따라서 $QTime_{gpu}$의 정확도를 높이기 위해 L_i^{id}의 종료시점을 파악해야 한다. 이를 위해, 우리의 프레임워크에서는 GPU-큐에서 대기중인 L_i^{id}에 대응되는 이벤트 커널인 E_i^{id}를 이용하여 이벤트 커널의 실행이 완료되었는지 확인한다. ([그림 4-5], 라인 26~28) E_i^{id}가 완료되면 $QTime_{gpu}$에서 T_i^{id}를 차감하여 업데이트 한다.([그림 4-5], 라인 29) 본 알고리즘을 구현하기 위해 cudaEvent를 사용하며 이벤트 커널의 실행과 실행 여부를 확인하기 위해 각각 *cudaEventRecord*와 *cudaEventQuery*를 사용했다. 일반적인 GPU 커널 실행과 달리 이벤트 커



[그림 4-7] 스케쥴링 방식에 따른 다중 DNN 실행 비교

널은 무시할수 있는 작은 시간동안 실행되기 때문에 [그림 4-6]에서 언급하지 않는다.

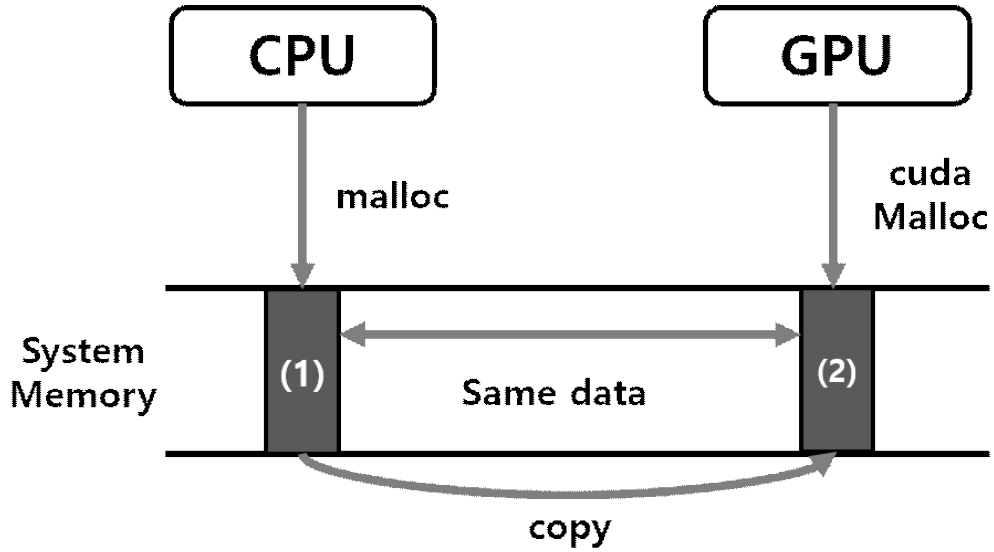
3) 최대한의 병렬 효과 얻기

2장에서 언급한 것과 같이 MPS는 타겟 디바이스에서 지원하지 않는다. 따라서 병렬처리를 위해 CUDA streams을 사용한다. CUDA streams을 활용

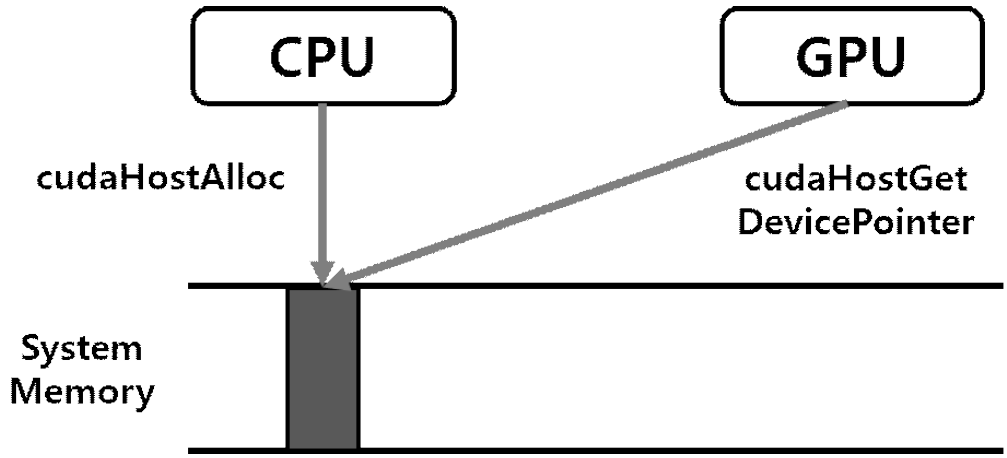
하면 CPU/GPU간의 데이터 전송 시간이 숨겨질 수 있고 앞에서 언급한 것과 같이 두 개의 레이어 작업이 서로 다른 CUDA streams에 있을 때 같이 실행할 수 있어 병렬처리 효과를 낼 수 있다. 그리고 커널 실행에 필요한 총 컴퓨팅 자원이 남는 한 다른 CUDA stream에 있는 여러개의 커널 함수가 같이 실행될 수 있다.(Amert, T., et al, 2017)

GPU의 병렬처리 성능을 높이는것 외에도 사용할 수 있는 CPU를 사용하여 총 실행시간을 줄일 수 있다. [그림 4-7]은 4가지의 스케쥴링 방법을 이용하여 DenseNet-201 3개를 동시에 실행할 때의 그림이다. 그림의 모든 레이어는 컨볼루션 레이어이고 L_j^2만 dense conn. 레이어이다. 그림에 표시된 숫자는 레이어의 실행시간이며 단위는 마이크로초 이다.

그림의 (a)와 (b)는 모든 레이어 작업을 GPU에서 실행하지만 (c)와 (d)는 CPU를 같이 이용한 효과를 보여준다. (a)는 DNN을 각 프로세스에 할당하여 컨텍스트 교환 오버헤드가 있는 경우이다. 반면에 (b)는 DNN들을 한 프로세스의 쓰레드로 관리하여 컨텍스트 교환 오버헤드를 제거한 경우이다. L_j^2는 그림에서 나타낸 것과 같이 CPU에서 실행시간은 390μs, GPU에서 실행시간은 100μs이다. (b)와 (c)를 비교하면 L_j^2을 CPU에서 실행하여 총 실행시간을 약 100μs만큼 단축할 수 있다. (c)에서는 기본 스트림만을 사용했기 때문에 L_{j-1}^2의 실행이 끝났음에도 불구하고 바로 전에 실행한 L_{k-1}^3 실행이 끝나면서 동기화되어야 하기 때문에 지연시간이 발생한다. 이 동기화 지연시간은 회색상자로 표시되어있다. (d)는 각 DNN모델을 서로 다른 CUDA streams에 할당한 예시를 나타낸다. CUDA streams은 스트림 단위로 동기화를 제공한다. (c)에서 이전 레이어 작업인 L_{j-1}^2이 끝났음에도 불구하고 L_j^2을 바로 실행할 수 없었던 것과 반대로 스트림 단위 동기화를 이용하면 동기화 지연시간이 (d)와 같이 줄어든다. (a),(b),(c),(d)를 비교하면 총 실행시간은 t_a에서 t_d로 단축된다. (d)에 표시된 타원은 CUDA Streams의 병렬 처리 효과를 나타낸다. 이는 그 시점의 GPU사용량에 따라 결정된다.



[그림 4-8] Xavier의 CPU-GPU간 메모리 할당 및 데이터 복사



[그림 4-9] Xavier의 제로카피

4) 서로 다른 코어의 동기화 및 데이터 전송 오버헤드 최소화

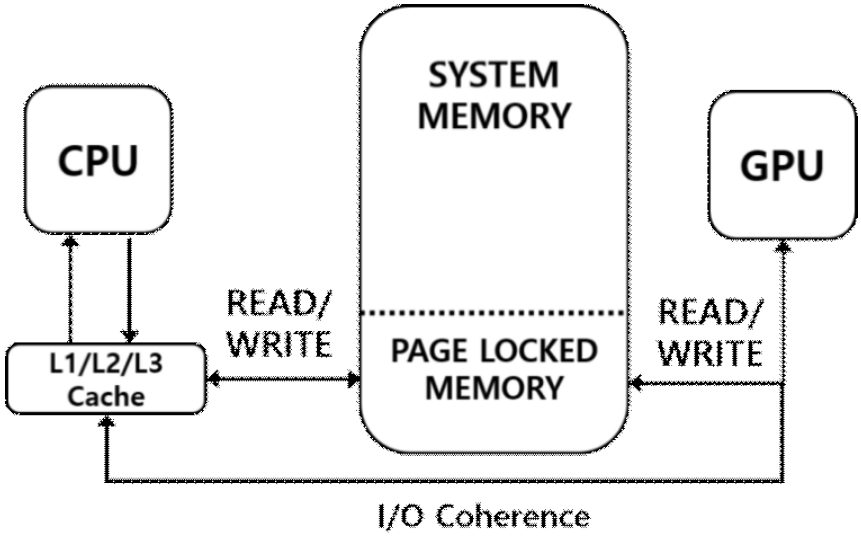
연속되는 두 레이어 작업이 실행되는 코어에 따라 다음과 같이 표현할 수 있다. $(C_{prev}, C_{curr}) \in \{(C,C),(C,G),(G,C),(G,G)\}$ 여기서 C_{prev}는 실행이 끝난 이전 레이어의 실행 코어 타입이며 C_{curr}은 다음에 실행될 레이어의 코어 타입이다. 그리고 C는 CPU를 G는 GPU를 나타낸다. (C,C)와 (G,G)의 경우 연속된 두 레이어 작업이 같은 코어에서 실행되기 때문에 데

이터 전송과 동기화를 위한 추가작업이 필요 없다.

하지만 (C, G)와 (G, C)는 연속되는 레이어가 서로 다른 코어에서 실행됨으로 데이터 전송이 필요하며 이는 큰 오버헤드를 발생시킨다. [그림 4-8]은 일반적인 CPU와 GPU의 데이터 복사를 보여준다. 그림에서 두 코어가 데이터를 사용하기 위해 각 코어의 메모리 공간에 데이터를 할당한다. 그런데 우리의 타겟 디바이스인 Xavier에서는 CPU와 GPU가 시스템 메모리를 물리적으로 공유하기 때문에 같은 메모리에 데이터가 중복 생성되는 문제가 발생한다. 우리는 이를 해결하기 위해 CUDA zero-copy를 이용한다.[13] 제로카피를 사용하기 위해서는 호스트(CPU)에 고정된 메모리(pinned memory)공간이 필요하다. 매핑된 고정 메모리는 추가적인 데이터 복사 없이 GPU가 CPU의 메모리 공간에 접근이 가능해진다. 제로카피는 CUDA runtime에서 제공하는 *cudaHostGetDevicePointer* 함수를 이용하여 쉽게 구현이 가능하다. 일반적으로 서버시스템에서 사용하는 d-GPU의 경우 제로카피를 적용했을 때 GPU는 PCIe를 통해 매번 호스트 데이터에 액세스하므로 데이터 접근 횟수가 늘어날수록 큰 오버헤드를 발생시킨다. 하지만 Xavier와 같은 i-GPU에서는 호스트와 디바이스가 [그림 4-9]와 같이 같은 물리적 메모리 공간을 사용하므로 직접 접근할 수 있기 때문에 제로카피의 효과가 더욱 크다. 그리고 GPU를 위한 추가 메모리 공간이 필요하지 않기 때문에 메모리 사용량을 줄일 수 있다.

(C, G)의 경우 CPU는 이전 레이어 작업이 완료된 이후에 다음 레이어 작업 커널을 발행한다. 따라서 추가 동기화 작업이 필요 하지 않다. 하지만 (G, C)의 경우에는 레이어 작업의 순서가 보장되지 않기 때문에 동기화를 위한 작업이 필요하다. L_i^{id}가 GPU에서 실행이 끝나기 전에 CPU에서는 L_{i+1}^{id}을 실행할 수 없다. 이 때 연속되는 레이어 작업을 순차적으로 실행할 수 있게 동기화를 해주어야 하는데 여기에는 *cudaDeviceSynchronize*를 이용한 디바이스 동기화와 *cudaStreamSynchronize*를 이용한 스트림 동기화가 있다. 앞에서 설명한것과 같이 스트림 동기화의 오버헤드가 더 적기 때문에 우리는 스트림 동기화를 이용한다. 그래서 제안된 프레임워크는 동일한 DNN의 모

13) Jetson-Zero-Copy, online: https://www.fastcompression.com/blog/jetson-zero-copy.htm



[그림 4-10] Xavier의 캐시 일관성

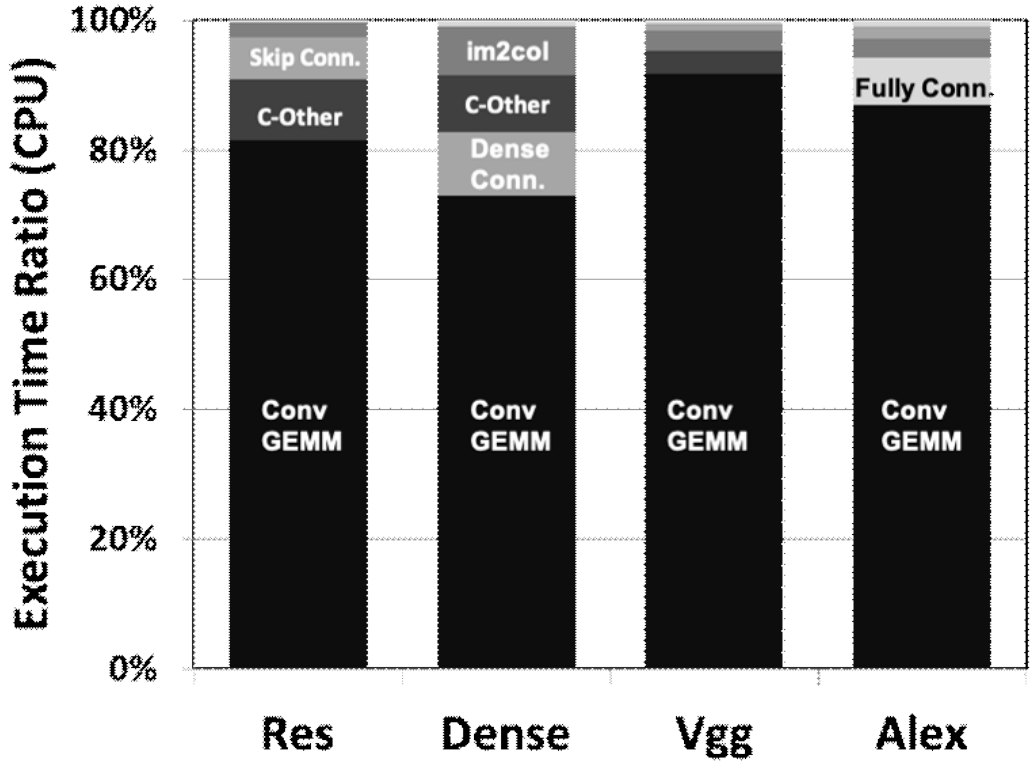
든 레이어를 같은 스트림에 할당한다. *cudaStreamSynchronize*를 이용하면 GPU에서 L_i^{id}이 실행이 끝나고 스트림에 다른 DNN의 레이어가 존재하지 않기 때문에 데이터를 바로 동기화 할 수 있으며 CPU에서 L_{i+1}^{id}를 바로 실행할 수 있다.

CPU-GPU간의 데이터 문제에서 더 중요한 것은 두 코어간의 캐시일관성(cache-coherence)문제이다. Xavier에서 CPU와 GPU간의 캐시 일관성은 호스트의 캐시를 통해 *I/O Coherence*라는 하드웨어 기술로 유지된다. 두 코어간 캐시 일관성을 통해 DRAM의 사용량을 줄일 수 있다.[14)]

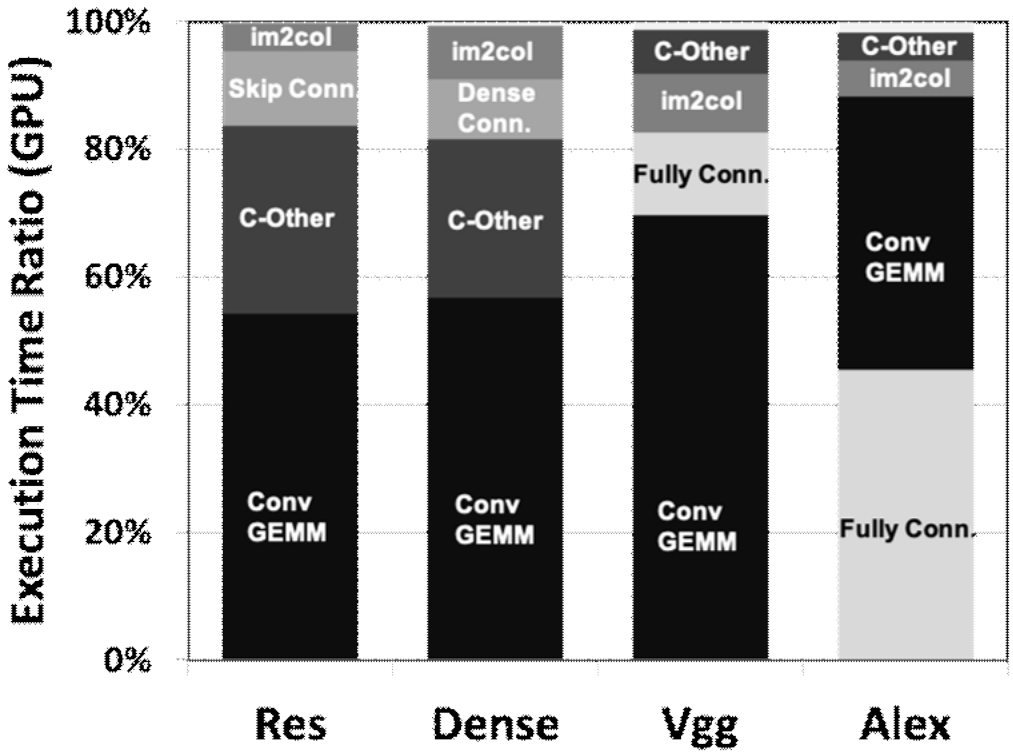
5) 레이어 작업의 실행시간 예측

제안된 프레임워크의 목표는 CPU-GPU Co-Scheduling을 통해 다중 DNN 실행의 총 실행시간을 줄이는 것이다. 따라서 레이어 작업을 위한 코어 선택기가 실행될 때마다 실행시간을 예측하는 것은 성능에 큰 영향을 줄 수 있다. 빠르고 정확한 실행시간 예측을 위해 높은 정확도를 보여준 선형회귀(Linear Regression)(Seber, G., et al, 2012)과 KNN(K-nearest neighbor)(James, G., et al, 2013) 두가지 모델을 선택하였다. GPU의 병렬

14) Nvidia Cuda Memory Management, online:
https://developer.ridgerun.com/wiki/index.php?title=NVIDIA_CUDA_Memory_Management



[그림 4-11] DNN 모델 각 레이어의 실행 시간 비율(CPU)



[그림 4-12] DNN 모델 각 레이어의 실행 시간 비율(GPU)

처리 특성 때문에 레이어 작업의 크기와 실행시간이 비선형적인 관계에 있다고 판단하여 KNN에 주목했다. 하지만 많은 DNN의 모든 레이어에서 코어 선택기를 실행할 때 예측 실행시간을 얻기 위해 데이
터의 모든 유클리드 거리를 계산하는 것은 상당한 오버헤드를 발생시키며 이는 프레임워크의 성능이점을 상쇄시킬 수 있다. 따라서 빠른 예측이 가능한 선형회귀모델을 사용하여 실행시간을 예측한다.

[그림 4-11]과 [그림 4-12]은 CPU와 GPU에서 DNN을 실행하여 전체 실행시간중 각 레이어가 차지하는 비율을 보여준다. 보다 정확한 예측을 위해 컨볼루션 레이어를 GEMM(general matrix to matrix multiplication), im2col, C-other로 분류한다. 단순하게 컨볼루션 레이어의 핵심 연산인 GEMM과 im2col을 제외한 데이터 초기화 및 활성함수 적용 등을 C-other라 한다.(Chetlur, S., et al, 2014; Sze, V., et al, 2017) [그림 4-11]과 같이 GEMM은 대부분의 연산이 MAC(multiply and accumulate)로 이루어져 있기 때문에 모델 연산의 대부분을 차지한다. 하지만 GPU의 경우 [그림 4-12]과 같이 GEMM이 차지하는 비율은 CPU보다 작다. GPU는 SIMT를 사용하여 GEMM의 MAC연산을 병렬로 처리할 수 있기 때문이다. 또한 GEMM, im2col, C-other은 사용하는 DNN모델마다 구성이 달라지기 때문에 세개의 예측모델로 나누어 더 정확한 실행시간을 예측할 수 있도록 한다.

$$E_i = w_1 x_{i1} + w_2 x_{i2} + \cdots + w_f x_{if} + b \tag{6}$$

실행시간 예측을 위해 [표 2-1]에 나열된 각 레이어의 특징을 선택하여 선형회귀 모델을 적용하였으며 각 레이어에 대응하는 선형회귀 모델이 존재한다. 선형회귀 모델은 식 (6)과 같이 나타낼 수 있으며 여기서 w_i는 해당 특징에 대한 가중치 x_i는 레이어 작업의 특징이다. 식 (6)의 출력은 각 레이어의 예측 실행시간이며 모든 선형회귀 모델은 CPU와 GPU 모두에 대한 예측 실행 시간을 계산한다.

기본적으로 우리는 각각의 레이어를 코어에 할당시키며 컨볼루션레이어는 GEMM, im2col, C-other의 예측 결과를 합하여 컨볼루션 레이어의 예측 실

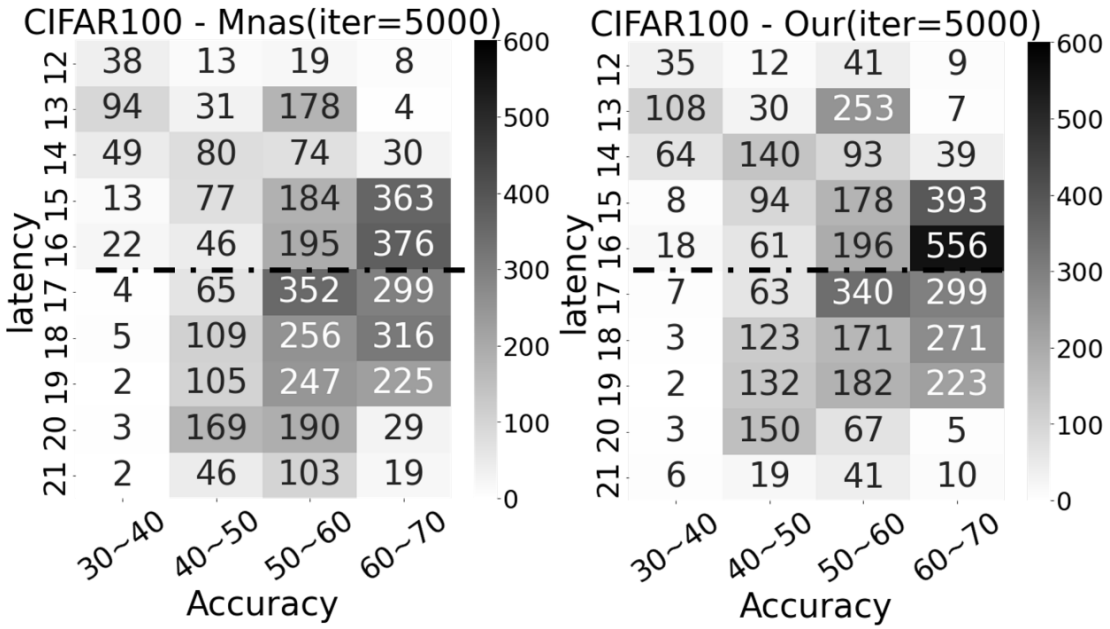
행시간으로 사용한다. 이를 통해 컨볼루션 레이어의 예측실행시간에 대한 선형회귀 모델의 정확도를 높일 수 있었다. 실행시간 예측을 위해 오프라인에서 DNN을 직접 실행하여 각 레이어의 실행시간 데이터를 수집하였다. 그리고 [표 2-1]의 다른 레이어들과 다르게 softmax 레이어와 average pooling 레이어의 갯수가 상대적으로 많이 적다. 그리고 이 레이어들은 CPU와 GPU 모두에서 실행시간이 대략 1μs정도이기 때문에 예측모델에서 제외하였다. 두 레이어는 항상 CPU에서 실행하도록 설정하였다.

d-GPU의 경우 (C, G)와 (G, C)에서 데이터 전송이 필요하기 때문에 데이터 전송 시간을 예측 실행시간에 추가해야 한다. 하지만 우리가 사용하는 제로카피와 스트림 동기화를 통해 (C, G)와 (G, C)에서의 데이터 전송시간은 매우 작다. 따라서 데이터 전송시간을 위한 예측 모델을 생성하지 않는다.

제 5 장 실험 결과 및 분석

본 장에서는 3장과 4장에서 제안한 연산 최적화 기법들의 효과를 실험으로 확인한다. 처음으로 3장에서 제안한 다중목적 보상함수가 적용된 NAS를 통해 빠른 탐색의 효과를 확인한다. 그리고 4장에서 제안한 다중 DNN 실행 프레임워크의 결과를 확인한다.

제 1 절 NAS 실험 결과



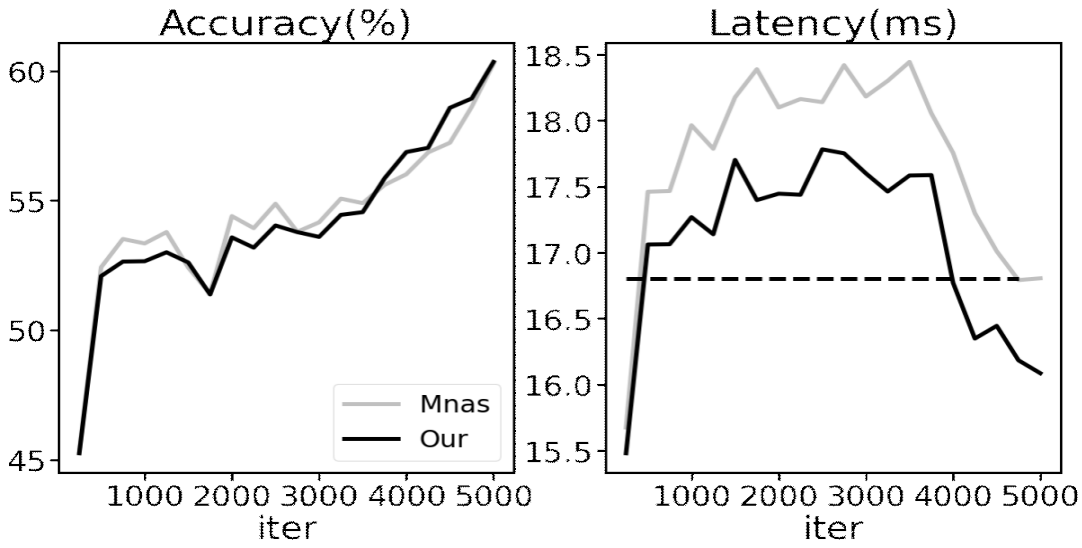
[그림 5-1] 컨트롤러가 생성한 DNN의 정확도와 실행시간의 히트맵 분석

실험은 MnasNet에서 사용한 보상함수와 본 논문에서 제안된 보상함수를 이용하며 나머지 NAS 시스템은 같은 시스템을 사용한다. 각 보상함수가 생성하는 DNN의 정확도와 추론 실행시간의 차이를 비교한다. MnasNet의 w값은 참고문헌 (Tan, M., et al, 2019) 에서와 같이 -0.07로 고정한다. 컨트롤러는 총 5,000번 DNN을 생성하며 학습된다. 추론 실행시간 제한조건인

T=17ms로 고정하며 NAS-Bench-201의 CIFAR-100으로 평가한 데이터셋에서 정확도, 추론 실행시간, DNN의 탐색 공간을 사용한다.

[그림 5-1]은 5,000번의 학습 동안 생성되는 DNN의 정확도와 실행시간을 이용한 히트맵 분석이다. 히트맵의 세로축은 소수점을 버린 값들이다. 즉 17.00 ~ 17.99 사이의 값은 모두 17에 포함된다. [그림 5-1]의 점선은 추론 실행시간 제한조건인 17ms를 기준으로 나눈것이며 점선의 위는 추론 실행시간이 17ms이하 이고 점선의 아래는 추론 실행시간이 17ms를 넘는 것들이다. MnasNet의 보상함수를 이용한 NAS 결과를 보면 MnasNet은 추론 실행시간의 제한조건을 엄격하게 제한하지 않기 때문에 17ms를 기준으로 넓은 범위의 DNN을 탐색한다. 하지만 제안된 보상함수를 사용한 NAS 결과에서는 추론 실행시간이 17ms 이하인 DNN을 더 많이 생성한다. 히트맵에서 추론 실행시간이 17ms보다 작으면서 정확도가 50~60%인 DNN은 MnasNet의 보상함수를 사용했을 때 659개, 제안된 보상함수를 사용했을 때 780개 이다. 그리고 똑같이 추론 실행시간이 17ms보다 작으면서 정확도는 60~70%인 DNN은 MnasNet의 보상함수를 사용했을 때 781개, 제안된 보상함수를 사용했을 때에는 1004개를 생성한 것을 확인할 수 있었다. 이를 통해 본 논문에서 제안한 보상함수는 제한조건을 엄격하게 지키기 때문에 탐색범위가 좁아지지만 제한조건을 지키는 것에만 집중하는 것이 아니라 정확도고 최대가 되는 DNN을 탐색할 수 있도록 컨트롤러를 학습시킬 수 있다는 것을 확인할 수 있다.

[그림 5-2]는 5,000번의 반복 동안 컨트롤러의 학습 상황을 볼 수 있게 컨트롤러가 생성하는 DNN의 정확도와 실행시간으로 평균을 낸 그래프이다. 총 5,000번의 반복을 20개의 구간으로 나누어 250개씩 평균을 냈다. [그림 5-2]의 정확도를 보면 두 개의 보상함수 모두 학습이 진행될수록 생성하는 DNN의 정확도는 높아지는 것을 알 수 있다. 이는 컨트롤러의 학습이 진행될 수록 정확도가 높은 DNN 위주로 생성한다는 것을 의미한다. 하지만 추론 실행시간의 그래프를 봤을 때 추론 실행시간 제한조건인 17ms 이하로 떨어지는 시점이 더 빠른 것으로 나타났다. 우리의 보상함수는 약 4,000번째 반복부터 17ms 이하로 떨어졌으며, MnasNet의 보상함수는 4,750번째 반복에



[그림 5-2] 컨트롤러 학습 동안 생성되는 DNN의 정확도 및 추론 실행시간

서 17ms 이하로 떨어진다. 이는 컨트롤러가 추론 실행시간이 17ms이하인 DNN 위주로 생성한다는 것을 의미한다. 즉, 제안된 다중목적 보상함수를 이용한 NAS는 기존보다 약 750번 덜 동작하더라도 제한조건 T보다 작은 추론 실행시간을 가지는 DNN 위주로 탐색할 수 있었다. 따라서 DNN모델 탐색을 더 빠르게 할 수 있기 때문에 전체 NAS 수행시간도 줄일 수 있다. 이는 제안된 보상함수가 제한조건 T를 넘는 경우에 보상을 크게 낮추기 때문에 T보다 작은 추론 실행시간을 가지는 DNN 위주로 탐색하게 되기 때문이다.

[표 5-1]은 특정 정확도와 추론 실행시간을 가지는 DNN이 있을 때 두 보상함수의 보상값을 비교하기 위해 결과를 나타낸 것이다. (a), (b), (c)는 각각의 비교군이며 각 경우에 DNN들은 다른 정확도와 추론 실행시간을 가진다. [표 5-1]의 괄호 안의 값은 (정확도, 실행시간) 의 순서로 적혀있다. (a)와 (b) 모두 5%의 정확도 상승 시 실행시간이 두 배가 되는 관계를 가지는 DNN들이다. 이는 MnasNet의 다중목적 보상함수의 정책을 따르는 관계이다. 여기서 (a)의 DNN 2는 추론 실행시간이 T보다 작은 경우이며 (b)의 DNN 2는 추론 실행시간이 T보다 큰 경우이다. (a), (b) 모두 MnasNet의 5%의 정확도가 상승할 때 추론 실행시간은 두 배가 되는 관계를 가지기 때문에 MnasNet의 보상함수를 사용했을 때에는 비슷한 보상값을 얻을 수 있었다. 하지만 제안된 보상함수는 (a)에서 DNN 1보다 DNN 2의 보상이 더 크

(a)	DNN 1 (80%, 8ms)	DNN 2 (84%, 16ms)
MnasNet	0.84334	0.84352
Proposed	0.82102	0.83546
(b)	DNN 1 (80%, 12ms)	DNN 2 (84%, 24ms)
MnasNet	0.81974	0.81996
Proposed	0.81186	0.77031
(c)	DNN 1 (80%, 8ms)	DNN 2 (83%, 16ms)
MnasNet	0.843344	0.833527
Proposed	0.82102	0.826460

[표 5-1] DNN의 정확도, 실행시간이 주어질 때 각 보상함수의 보상값 비교

고 (b)에서는 DNN 1보다 DNN2의 보상이 더 작은 것을 확인할 수 있다. (a)는 두 DNN의 추론 실행시간이 모두 T보다 작기 때문에 정확도가 더 높은 DNN 2에 더 큰 보상을 준다. 그리고 (b)의 DNN 2는 정확도가 더 높음에도 불구하고 추론 실행시간이 T보다 크기 때문에 더 작은 보상을 주는 것을 확인할 수 있다. (c)에서는 DNN 1과 DNN 2의 추론 실행시간은 두 배 차이나지만 정확도는 5%이하로 증가한 경우를 나타낸다. 따라서 MnasNet의 보상함수로 계산했을 때에는 DNN 1이, 제안된 보상함수로 계산했을 때에는 DNN 2가 더 큰 보상을 가진다. DNN 2가 DNN 1보다 정확도가 높으며 DNN 2도 추론 실행시간이 T보다 작기 때문에 목표로 하는 DNN에 더 적합하다. 따라서 제안된 보상함수가 제대로 동작하는 것을 확인할 수 있다.

제 2 절 다중 DNN 실행 결과

실험에 사용한 타겟 디바이스 Xavier의 하드웨어 상세 정보는 [표 5-2]와 같다. 본 절에서는 4장에서 제안한 다중 DNN 실행을 위한 다양한 조건에서의 실행시간 비교와 스케쥴링의 효과를 확인한다. 실험에는 총 4개의 DNN을 사용한다. 4개의 DNN은 DenseNet-201, ResNet-152, VGG-16,

CPU	8-Core ARM v8.2 64-Bit CPU, 8MB L2, 4MB L3 Cache
GPU	512-core Volta GPU with 64 Tensor cores 11 TFLOPs (FP16), 22 TOPS(INT8)
Memory	32GB 256-bit LPDDR4x 2133MHZ - 137GB/s
OS	Linux kernel version 4.9.140-tegra
CUDA version	CUDA v10.0.166
JetPack	v.4.2 [L4T 32.1.0]

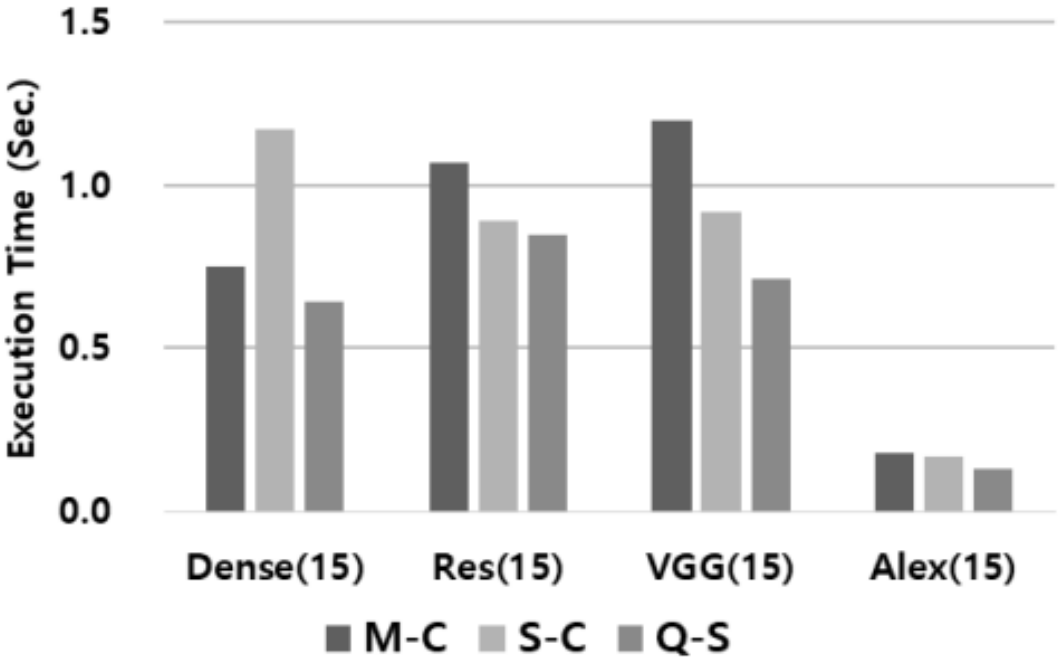
[표 5-2] 타겟 디바이스 Jetson AGX Xavier의 하드웨어 상세 정보

AlexNet이다. 먼저 다중 DNN 실행 시, GPU의 병렬 처리 성능을 최대로 끌어올렸을 때의 효과를 확인하고 그 후에 CPU-GPU Co-Scheduling의 효과를 확인한다.

1) 다중 DNN 실행 시 GPU 오버헤드 최소화 결과

가) 같은 종류의 DNN 다중 실행

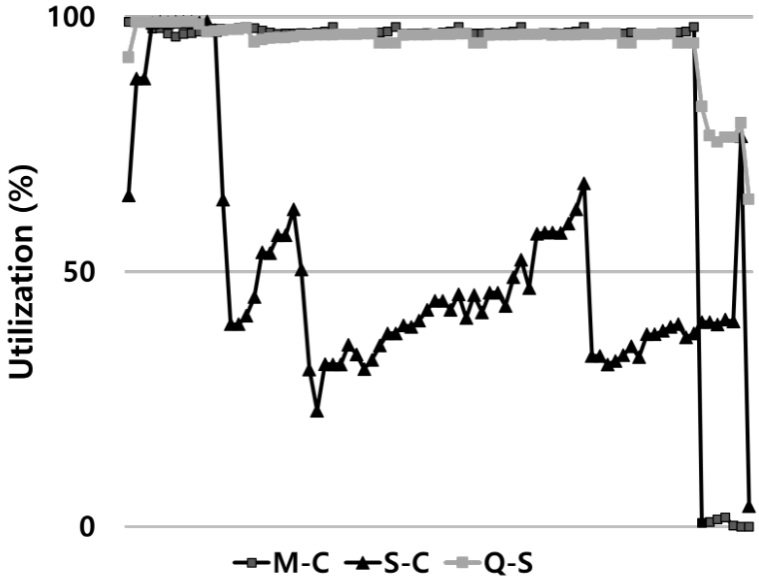
본 실험에서는 동일 종류의 DNN 모델을 동시에 실행했을 때 세 가지 환경에서의 실행시간을 비교하여 효과를 확인한다. [그림 5-3]은 세 가지 환경에서 각각 같은 종류 DNN 15개를 동시에 실행시켰을 때 총 실행시간을 비교한다. M-C는 멀티 컨텍스트 환경, S-C는 싱글 컨텍스트 환경, Q-S는 4장에서 제안한 GPU 오버헤드를 최소화한 프레임워크를 나타낸다. [그림 5-3]의 Dense(15)는 DenseNet-201 15개를 동시에 실행시킨 것을 의미하며 총 실행시간은 15개의 DNN 연산이 모두 끝날 때 까지의 시간을 의미한다. [그림 5-3]의 결과를 보면 Res, VGG, Alex 모두 M-C의 실행시간이 가장 크며 제안된 프레임워크인 Q-S의 실행시간이 가장 작은 것을 확인할 수 있다. Res(15)에서는 Q-S를 기준으로 M-C보다 21% 감소하였으며 S-C보다 4.8% 감소하였다. VGG(15)는 M-C보다 40.8%, S-C보다 22.8% 감소하였으며 Alex(15)는 M-C보다 26.5%, S-C보다 22.6% 감소한 것을 확인할 수



[그림 5-3] GPU 실행환경에 따른 같은 종류 DNN들의 실행 시간

있었다.

Dense(15)에서도 Q-S를 기준으로 M-C보다 14.7%, S-C보다 45%가 감소하였고 Q-S의 실행시간이 가장 작은 것은 다른 DNN들과 동일하지만, 특이하게 M-C보다 S-C에서의 실행시간이 더 길게 나타난다. 이는 컨텍스트 교환 오버헤드보다 싱글 컨텍스트의 mutex 오버헤드의 영향을 더 크게 받은 것으로 볼 수 있다. [그림 5-4]는 Dense(15)를 각 환경에서 실행했을 때의 시간별 GPU 사용량을 나타낸 그림이다. x축은 총 실행시간을 비율로 0% 에서 100%까지 표현한다. 그리고 y축은 각 구간의 GPU 사용량을 나타낸다. [그림 5-4]에서 M-C와 Q-S는 실행시간 대부분이 GPU를 100%에 가깝게 사용하지만 S-C의 경우 20%를 실행한 구간부터 GPU 사용량이 50% 이하로 떨어진다. 이는 DenseNet-201의 특성 상 실험에 사용한 다른 DNN 보다 레이어의 수는 가장 많으며 레이어당 실행시간은 짧기 때문에 mutex를 거는 횟수가 더욱 많아지며 S-C의 mutex 잠금/획득으로 인한 오버헤드로 GPU를 원활하게 사용할 수 없고 유휴시간이 발생했기 때문에 이와 같은 결과가 나온 것을 확인할 수 있었다.



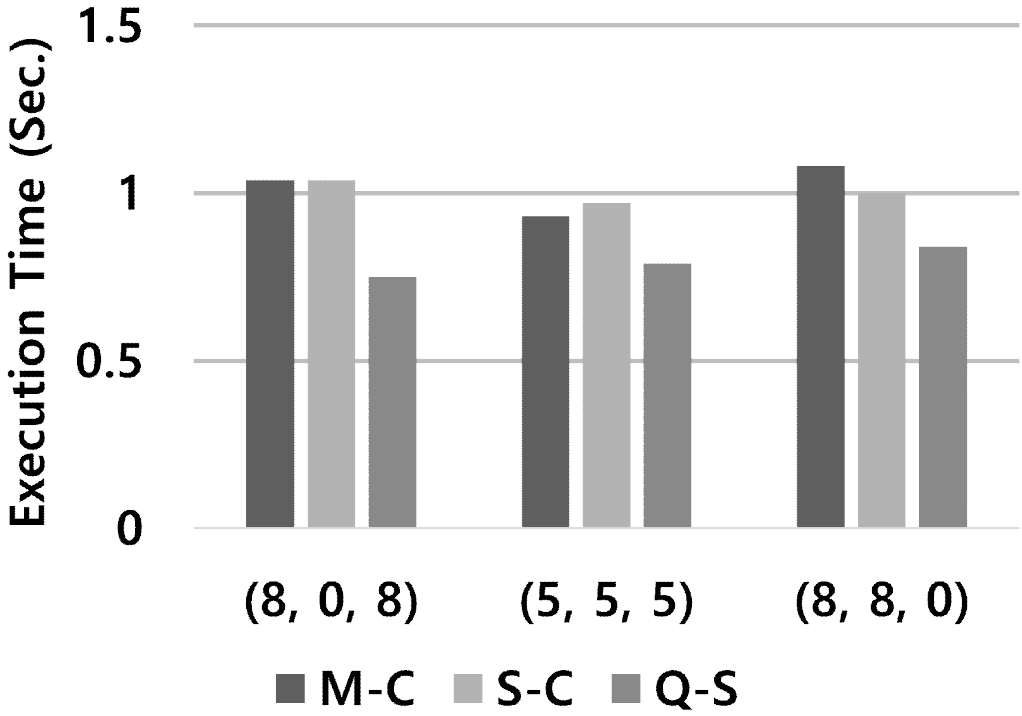
[그림 5-4] Dense(15)의 시간에 따른 GPU 사용량

나) 다른 종류의 DNN 다중 실행

여기서는 다른 종류의 DNN을 동시에 실행한 결과를 비교한다. 앞의 실험과 달리 다른 종류의 DNN은 총 실행시간이 각각 다르기 때문에 조금 다른 결과를 확인할 수 있다. 실행시간이 가장 짧은 Alex는 제외하며 나머지 세 개의 DNN을 (Dense, Res, VGG)의 순서로 배치한다. [그림 5-5]는 실험 결과를 보여주며 (8, 0, 8)은 Dense 8개, Res 0개 VGG 8개를 표현한다.

실험 결과를 보면 (8, 0, 8)에서는 M-C와 S-C의 실행시간이 비슷하며 Q-S는 약 27.8% 감소하였다. (5, 5, 5)에서 Q-S는 M-C보다 15%, S-C보다 18.5% 감소하였으며 (8, 8, 0)에서는 M-C보다 22%, S-C보다 16% 감소한 것을 확인할 수 있다.

실험 결과로 같은 종류의 DNN 다중 실행과 다른 종류의 DNN 다중 실행 모두 M-C와 S-C 보다 제안된 Q-S의 실행시간이 가장 짧은 것을 확인할 수 있었다. 그리고 각각의 DNN이 M-C와 S-C에 영향을 받는 정도가 다른 것을 확인할 수 있었다.

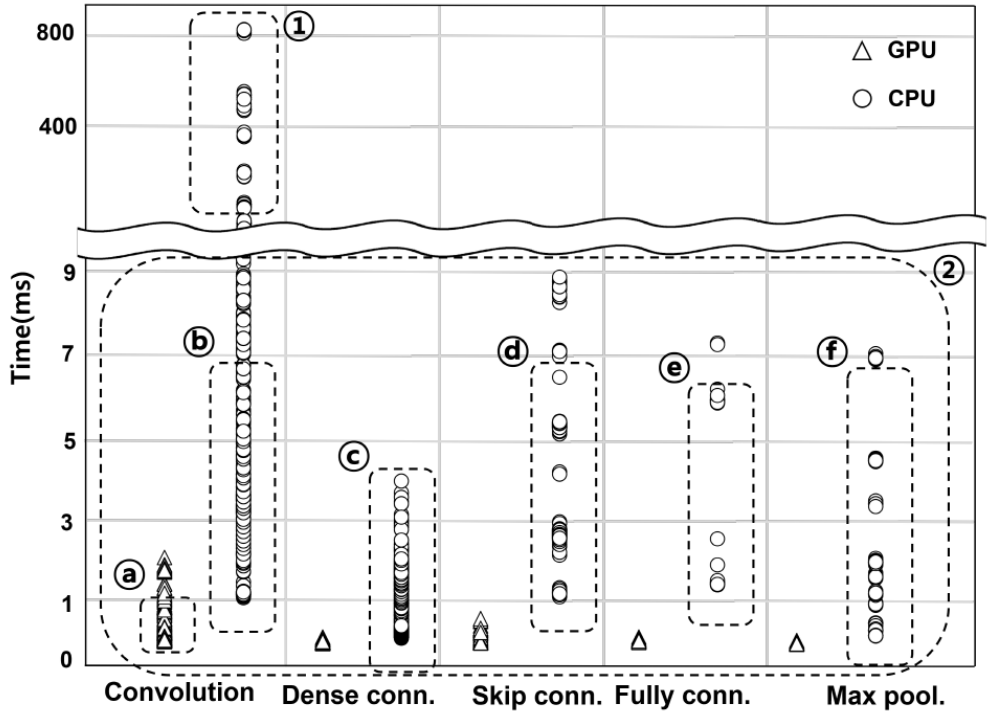


[그림 5-5] GPU 실행환경에 따른 다른 종류 DNN들의 실행 시간

2) 다중 DNN 실행을 위한 CPU-GPU Co-Scheduling 결과

가) 실행시간 예측을 위한 데이터 수집 결과

GPU의 부하를 줄이기 위해 CPU를 사용하는 것이 도움이 되는지 확인하기 위해 수집된 실행시간 데이터를 분석한다. 각 코어(CPU, GPU)에서 4개의 DNN 모델안의 모든 레이어의 실행시간을 수집한다. 데이터의 변동성을 줄이기 위해 실행시간 데이터 수집을 하는동안 코어의 성능을 고정하며 한번에 한개씩 실행하여 수집한다. [그림 5-6]은 실험에서 사용하는 DNN모델 각 레이어의 실행시간 분포를 나타낸다. [그림 5-6]의 ①은 CPU에서 연산량이 큰 컨볼루션 레이어가 실행되는 경우를 보여준다. CPU에서 많은 시간이 소요되는 레이어는 CPU에서 실행될 가능성이 거의 없다. 여기서 CPU에서 실행 가능성이 있는 레이어의 특징은 ②에 나타난다. 예를들어 현재 GPU-큐에 ⓐ와 실행시간이 비슷한 컨볼루션 레이어가 여러개 대기중일 때 GPU-큐에서 대기하는 레이어 작업의 실행시간 합이 7ms보다 큰 경우 ⓑ~ⓕ와 같은 레이어 작업을 CPU에서 실행하여 GPU만 사용하는 것 보다 더 좋은 성능을 이끌



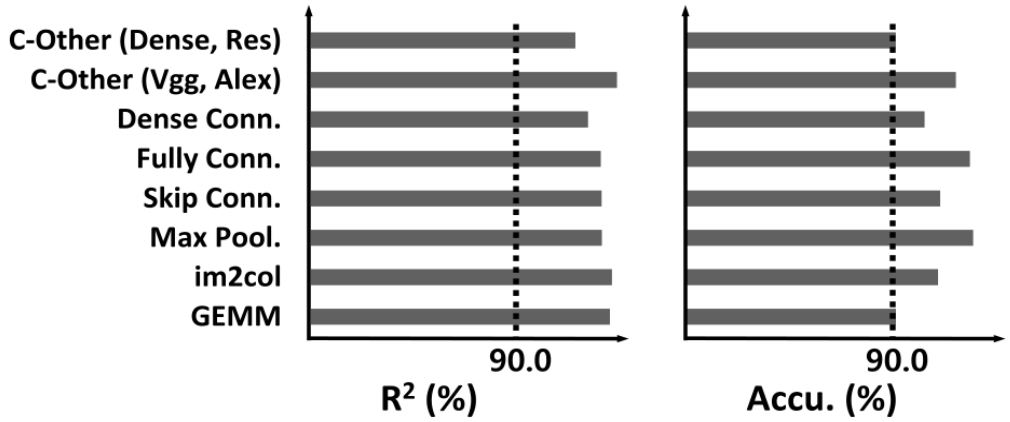
[그림 5-6] 4개의 DNN 모델의 모든 레이어에 대한 실행시간

어 낼 수 있다.

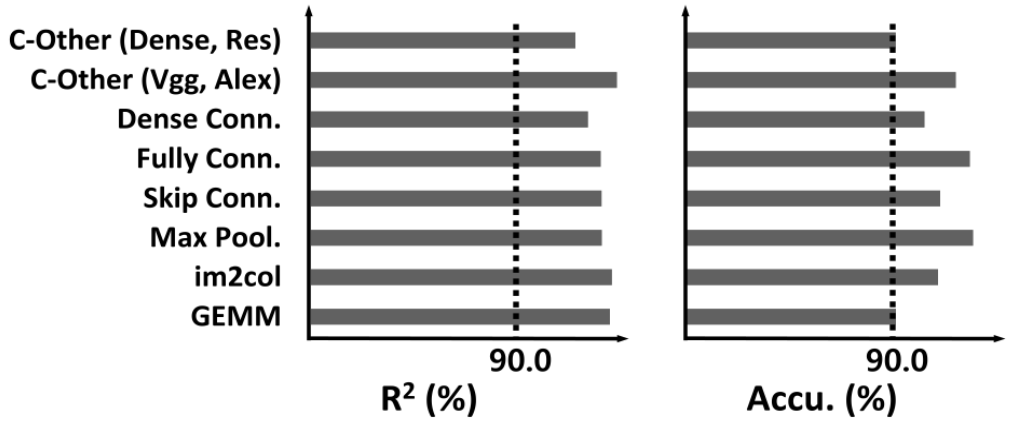
나) 실행시간 예측 모델의 정확성 검증

예측 모델로 사용한 선형회귀 모델의 성능을 확인하기 위해 결정계수(R^2)와 실행시간 예측 정확도를 이용한다. 결정계수는 만들어진 선형회귀 모델이 주어진 데이터를 얼마나 잘 표현하는지에 대한 성능 지표이며, 예측 정확도는 수집된 실행 시간과 예측한 실행시간의 차이가 허용 오차보다 작으면 참, 아니면 거짓으로 판단하여 평균을 낸 결과값이다. 허용 오차를 사용하는 이유는 실제 실행시에는 예측할 수 없는 변수가 많기 때문에 예측된 실행시간과 실제 실행시간이 정확하게 항상 같을 수 없기 때문이다. 수집된 데이터의 90%를 선형회귀 모델을 학습하기 위한 학습 데이터로 사용하고, 나머지 10%의 데이터를 이용하여 모델을 검증하였다.

[그림 5-7]과 [그림 5-8]은 각각 CPU와 GPU의 실행시간 예측 모델에 대한 검증 결과를 나타낸다. CPU와 GPU 모두 예측 모델의 결정계수 평균은



[그림 5-7] 각 레이어별 실행시간 예측 모델의 정확도 및 결정계수 - CPU

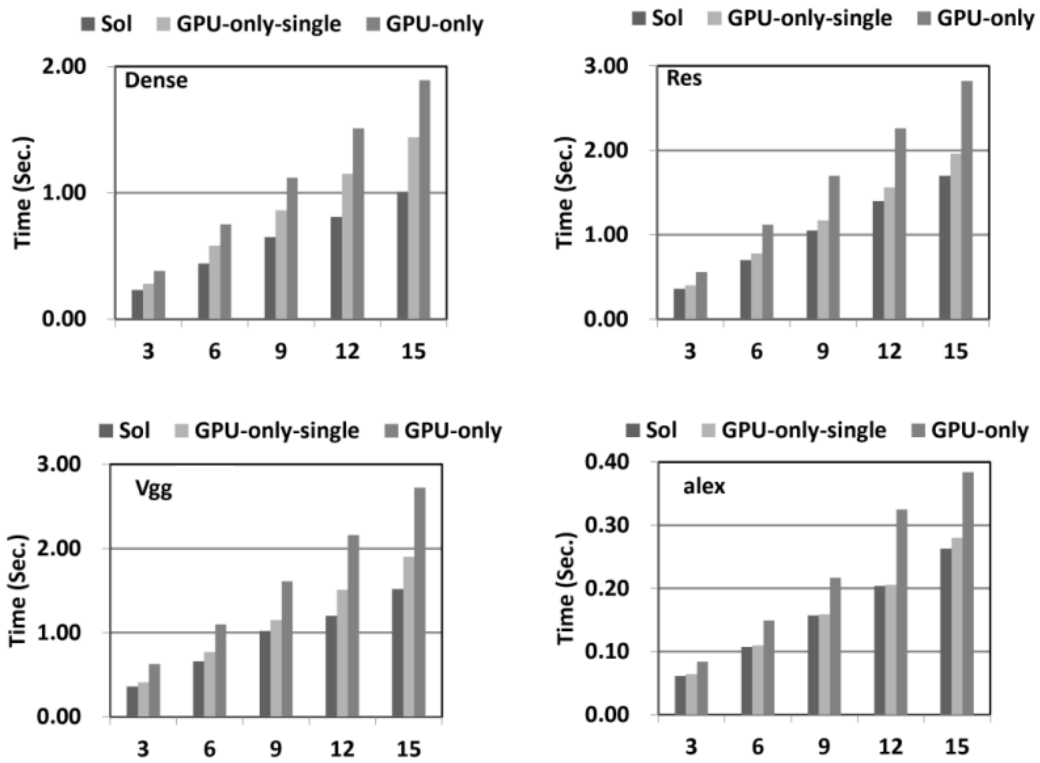


[그림 5-8] 각 레이어별 실행시간 예측 모델의 정확도 및 결정계수 - GPU

각각 95.2%, 96.2%이다. 이는 선형회귀 모델이 수집된 데이터를 잘 표현하고 있다고 말할 수 있다. 그리고 CPU와 GPU 예측모델의 정확도 평균은 각각 94.6%, 98.3%이다. 이는 각 레이어 실행시간을 선형회귀 모델로 충분히 예측할 수 있다는 것을 의미한다.

다) 같은 종류의 DNN 다중 실행

CPU-GPU Co-Scheduling의 효과를 확인하기 위해 같은 종류 DNN 다중 실행을 세 가지 환경에서 비교한다. GPU-only는 각 DNN 모델을 프로세스에 할당하여 실행한 멀티 프로세스 기반의 실행 환경이다. GPU-only-single은 싱글 프로세스 멀티 쓰레드의 형태이며, Sol은 제안된



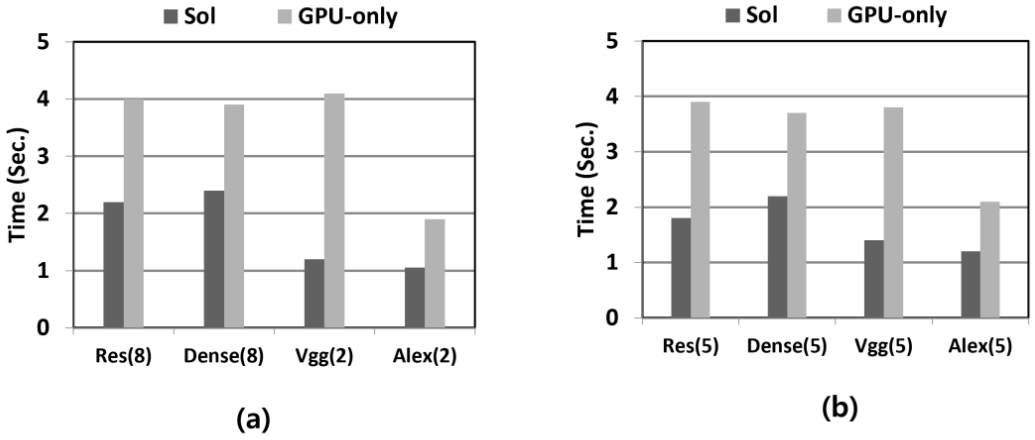
[그림 5-9] 세가지 스케쥴링 환경에 따른 실행시간 결과

CPU-GPU Co-Scheduling을 적용한 결과이다.

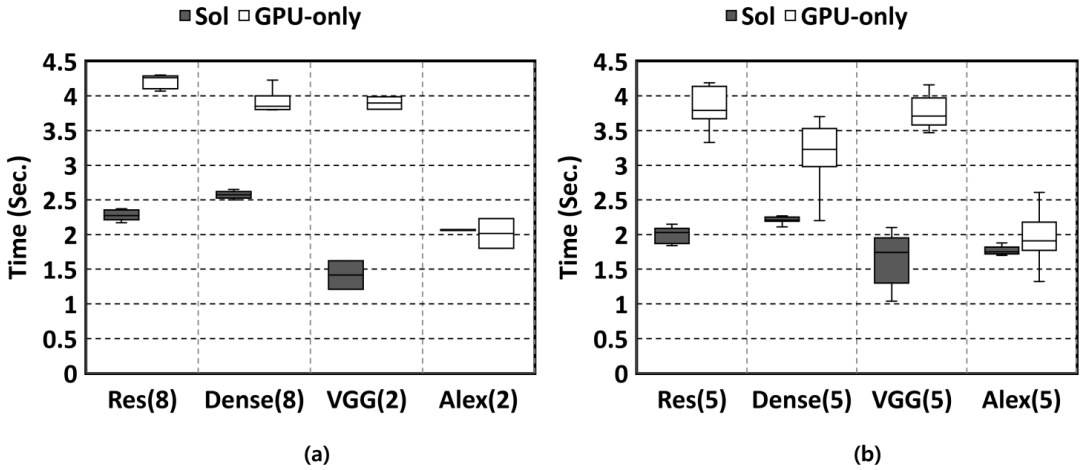
[그림 5-9]는 세가지 스케쥴링 환경에 따른 같은 종류 DNN 다중 실행의 결과이다. x축은 동시에 실행되는 DNN의 갯수를 나타내며 y축은 실행된 모든 DNN이 종료될 때 까지 소요되는 실행시간을 나타낸다. [그림 5-9]의 세가지 환경의 실행시간 차이는 동시에 실행되는 DNN의 갯수가 많아질 수록 더 명확하게 늘어난다. DenseNet-201의 경우 GPU-only과 비교했을 때 Sol은 약 39.5%에서 46.6%의 성능이, ResNet-152의 경우 35.7%에서 39.7%의 성능이 향상되었다. VGG-16과 AlexNet의 경우 각각 42.8%~44.1%, 27.4%~31.5%의 성능이 향상되었다.

라) 다른 종류의 DNN 다중 실행

이번에는 4개의 DNN을 적절히 조합하여 다른 종류의 DNN을 동시에 실행하여 결과를 확인한다. [그림 5-10]의 (a)는 ResNet-152, DenseNet-201,



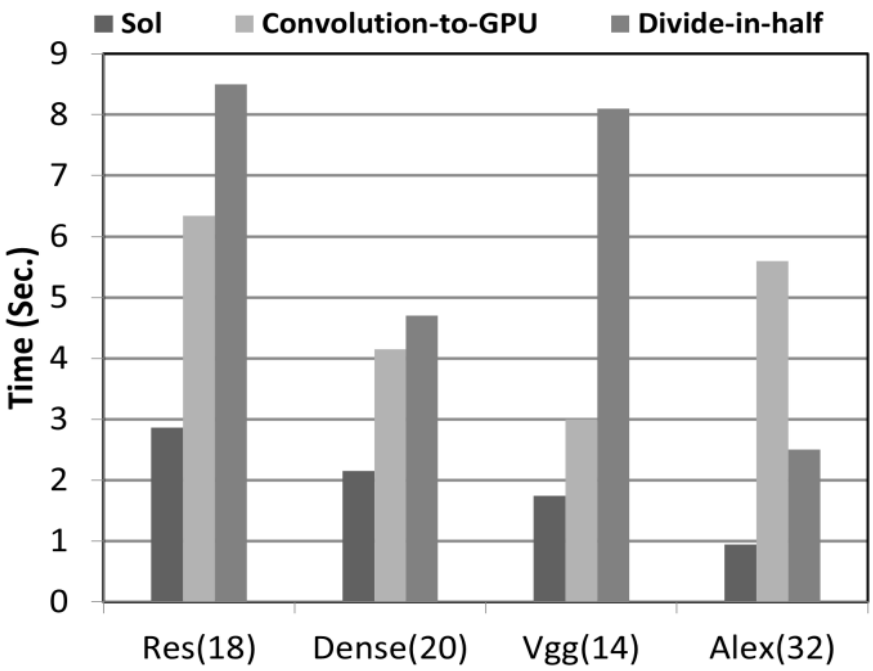
[그림 5-10] 두가지 스케쥴링 환경에서 서로 다른 종류 DNN 다중실행 결과



[그림 5-11] [그림 5-10]의 데이터 분포

VGG-16, AlexNet의 갯수를 각각 8, 8, 2, 2개로 설정한 결과이며 (b)는 모두 5개씩 설정한 결과이다. 두 실험 모두 20개의 DNN모델을 동시에 실행한다. x축의 괄호는 해당 DNN의 갯수를 나타내며 y축의 실행시간은 동일한 DNN모델이 전부 종료된 시간을 나타낸다. [그림 5-10]에 따르면 두가지의 실험 결과에서 모두 Sol의 실행시간이 38~70% 단축된다.

[그림 5-11]은 [그림 5-10]의 실행시간의 분포와 평균값을 확인하기 위해 상자수염 그래프로 표현한다. [그림 5-11]의 (b)의 결과를 보면 Sol의 가장



[그림 5-12] 코어 선택 정책에 따른 같은 종류 DNN 다중 실행 결과

큰 값과 가장 짧은 값의 차이는 각 DNN 모델에 대해 0.31초, 0.16초, 1.06초, 0.18초인 반면에 GPU-only의 경우 0.86초, 1.5초, 0.69초, 1.29초의 차이를 보인다. 그에 더해 평균 실행 시간도 각각 10.4%, 27.5%, 47.3%, 56.9%가 단축된다. 실험의 결과로 우리가 제안한 Sol이 DNN을 여러 개 실행할 때 더 균등한 실행시간을 가지도록 하며 평균 실행시간도 줄일 수 있다는 것을 알 수 있다.

앞의 실험 결과들로 미루어 보아 CPU-GPU Co-Scheduling 프레임워크를 통해 여러 개의 DNN을 동시에 실행할 때 갯수나 DNN의 종류에 관계없이 실행 성능을 높일 수 있다고 할 수 있다.

마) 동적 코어 선택 알고리즘의 효과

마지막으로 동적인 CPU-GPU 코어 선택의 효과를 확인하기 위해 정적으로 코어를 선택하는 두가지 예시 스케쥴링 알고리즘과 함께 비교한다. 첫번째로 Convolution-to-GPU는 모든 컨볼루션 레이어를 GPU에 할당하고 나머지 레이어는 모두 CPU에 할당하는 방식이다. 이 스케쥴링 방식은 CNN기반

DNN의 연산량중 약 90%이상이 컨볼루션 레이어 연산이라는 것에 근거한다.[47] 두번째 Divde-in-half는 N개의 DNN모델을 실행할 때 앞의 N/2개의 DNN을 CPU에서 N/2개의 DNN을 GPU에서 실행하는 경우이다.

[그림 5-12]에서는 앞에 언급한 두 정적 스케쥴링과 제안된 동적 스케쥴링을 비교한다. x축의 괄호 안의 숫자는 동시에 실행되는 DNN의 갯수 이며 y축은 모든 DNN의 실행이 끝나는 시간이다. 각각 ResNet-152는 18개, DenseNet-201은 20개, VGG-16은 14개, AlexNet은 32개 실행하였다. [그림 5-12]에서 확인할 수 있듯이 Sol의 성능이 가장 높았으며 42%에서 83%의 성능 차이가 난다. AlexNet을 제외하고 Convolution-to-GPU는 Divide-in-half에 비해 실행시간이 더 작다. 이는 컨볼루션 연산이 DNN연산의 대부분을 차지하며 Convolution-to-GPU에 비해 Divide-in-half에서는 더 적은 수의 컨볼루션 레이어를 GPU에서 실행하기 때문에 더 느리게 실행된다. 하지만 AlexNet(32)의 경우에는 Convolution-to-GPU는 32개의 GPU 컨텍스트를 사용하고, Divide-in-half는 16개의 컨텍스트를 사용하므로 컨텍스트 스위칭 오버헤드가 더 적다. 그리고 AlexNet의 컨볼루션 레이어의 연산은 다른 세 DNN 보다 연산량이 크지 않기 때문에 CPU사용에서의 이득이 더 크다. 따라서 AlexNet(32)의 Divide-in-half는 Convolution-to-GPU보다 더 빠르게 실행된다. 그렇지만 여기서도 제안된 스케쥴링 방식이 가장 빠른 것으로 나타난다. 이 실험 결과를 통해 동적 코어 선택 알고리즘을 통해 실행시간을 단축할 수 있다는 것을 확인할 수 있었다.

제 6 장 결론

본 논문에서는 엣지 디바이스에서 DNN 연산을 최적화 하기 위하여 서버리스 환경에서의 실행에 최적화된 DNN생성과 기존에 존재하는 높은 성능을 가진 DNN의 다중 실행을 최적화하는 두 가지 방법에 대해 분석하고 적절한 솔루션을 제안하였다.

첫번째로 NAS에 새로운 다중목적 보상함수를 적용하여 제한 조건에 맞는 DNN을 빠르게 탐색할 수 있었다. MnasNet의 접근 방법을 기반으로 제한 조건에 더 엄격한 다중 목적 보상함수를 제안하였고, 실험의 속도를 높이기 위해 Nas-Bench-201을 이용하였다. 제안된 보상함수의 목표는 실행시간의 제한 조건보다 빠르게 실행되는 DNN중 가장 정확도가 높은 DNN을 탐색하도록 하는 것이다. 그리고 실험으로 제한 조건보다 빠른 추론 실행시간을 가지면서 정확도가 가장 높은 DNN에 높은 보상을 준다는 것을 확인할 수 있었다. 그리고 이를 통해 탐색의 수렴속도가 더 빨라져 조건에 맞는 DNN을 더 빠르게 탐색하도록 할 수 있었다.

두번째로 제한된 리소스를 가지는 서버리스 환경에서의 다중 DNN 연산을 최적화 하는 방법을 제시하였다. 타겟 디바이스인 Xavier의 MPS 미지원으로 인해 GPU의 병렬처리 성능을 최대한으로 이끌어내기 위해 CUDA Streams를 사용하였다. 그리고 GPU의 부하를 줄이기 위해 CPU를 이용하였다. 그리고 CPU를 이용하기 위해 DNN의 레이어 연산 특성을 분석하고, CPU와 GPU에 동적인 레이어 작업 할당을 위해 선형회귀를 이용한 실행시간 예측 모델과 코어 선택 알고리즘을 제안하였다. 이를 통해 효과적으로 GPU에 집중되는 부하를 적절히 CPU에 분배할 수 있었고 실제로 실행시간이 최대 46.6% 향상된 것을 확인할 수 있었다.

1. 국외문헌

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.

Bojarski, M., Testa, D. D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Müller, U., Zhang, J., Zhang, X., Zhao, J., & Zieba, K. (2016) End to end learning for self-driving cars. arXiv preprint arXiv:1604.07316

Chen, Y., Luo, T., Liu, S., Zhang, S., He, L., Wnag, J., Li, L., Chen, T., Xu, Z., Sun, N., & Temam, O., (2014) DaDianNao: A machine-learning supercomputer,' in Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchitecture, pp. 609-622

Chen, Y.-H., Krishna, T., Emer, J. S. & Sze, V. (2017) Eyeriss: An energy efficient reconfigurable accelerator for deep convolutional neural networks. IEEE J. Solide-State Circuits. vol. 52, no. 1, 127-138

Cheng, H. T., Koc, L., Harmsen, J., Shaked, T., Chandra, T., Aradhey, H., Anderson, G., Corrado, G., Chai, W., Ispir, M., Anil, R., Haque, Z., Hong, L., Jain, V., Liu, X., & Shah, H. (2016) Wide & deep learning for recommender systems. in Proc. 1st workshop deep learn Recommender Syst, 7-10

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., & Shelhamer, E., (2014) CuDNN: Efficient primitives for deep learning arXiv preprint arXiv:1410.075

Chrabaszcz, P., Loshchiov, I., & Hutter, F. (2017) A downsampled variant of imagenet as analternative to the cifar datasets. arXiv preprint arXiv:1707.08819

Deb, K., (2014) Multi-objective Optimization. Search methodologies, 403-449

Deng, J., Dong, W., Socher, R., Li, L. J., Li, K., & Fei-Fei, L. (2009) magenet: A large-scale hierarchical image database. In CVPR

Dong, X., & Yang, Y. (2020) NAS-Bench-201: Extending the Scope of Reproducible Neural Architecture Search. in ICLR

Dyrstad, J. S. & Mathiassen, J. R. (2017) Grasping vitual fish: A step towards robotic deep learning from demonstration in virtual reality. (ROBIO), 1181-1187.

He, K., Zhang, X., Ren, S., & Sun, J. (2016) Deep Residual Learning for Image Recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition(CVPR)

He, X., Bowers, S., Candela, J. Q., Pan, J., Jin, O., SXu, T., Liu, B., Xu, T., Shi, Y., Atallah, A., & Herbrich, R. (2014) Practical lessons from predicting clicks on ads af Facebook. In Proceedings 20th ACM SIGKKD Conf. Knowl. Discovery Data Mining(ADKDD), 1-9

Higginbotham, S., (2016) Google takes unconventional route with homegrown machine learning chips. Next Platform, online: https://www.nextplatform.com/2016/05/19/google-takes-unconventional-route-homegrown-machine-learning-chips/

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wnag, W., Weyand, T., Andreetto, M., & Adam, H. (2017) MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv preprint arXiv:1704.04861

James, G., Witten, D., Hastie, T., & Tibshirani, R., (2013) An Introduction to Statistical Learning. Springer

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.

Kim, Y., Kim, J., Chae, D., Kim, D., & Kim, J., (2019) μLayer: Low latency on□device inference using cooperative single-layer acceleration and processor-friendly quantization," in Proc. 14th EuroSys Conf., 45

Krizhevsky, A., & Hinton, G. (2009) Learning multiple layers of features from tiny imagenet. Technical report

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

Li, X., Liang, Y., Yan, S., Jia, L., & Li, Y. (2019) A coordinated tiling and batching framework for efficient GEMM on GPUs," in Proc. 24th Symp. Princ. Pract. Parallel Program, 229-241

Lin, S.-C., Zhang, Y., Hsu, C.-H, Skach, M., Haque, M. E., Tang, L., & Mars, J. (2018). The architectural implications of autonomous driving: Constraints and acceleration. in Proc. 23rd Int. Conf. Architectural Support Program. Lang. Operating Syst., 751-766

Seber, G. A. F., & Lee, A. J., (2012) Linear Regression Analysis, Hoboken, vol. 936

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

Sze, V., Chen, Y.-H., Yang, T.-J., & Emer, J. S. (2017) Efficient processing of deep neural networks: A tutorial and survey. in Proc. IEEE vol. 105, no.12, 2295-2329

Tan, M., Chen, B., Pnag, R., Vasudevan, V., Sandler, M., Howard, A., & Le, Q. V. (2019) MnasNet: Platform-Aware Neural Architecture Search for Mobile. in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)

Vasisht, D., Kapetanovic, Z., Won, J. H., Jin, X., Chandra, R., Kapoor, A., Sinha, S. N., Sudarshan, M., & Stratman, S. (2017) Farmbeats: An IoT platform for data-driven agriculture. in Proc. NSDI, 515-529

Williams, R. J., (1992) Simple statistical gradient-following algorithms for connectionist reinforcement learning. In Machine Learning

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.

Zhang, X., Zhou, X., Lin, M., & Sun, J. (2018) ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)

Zhoh, B., Vasudevan, V., Shlens, J., & Le, Q. V. (2018) Learning Transferable Architectures for Scalable Image Recognition. in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)

Zops, B. & Le, Q. V. (2017) Neural Architecture Search with Reinforcement Learning in ICLR

ABSTRACT

Research on Optimized Computation Technique for Deep Neural Network in Cloud-Independent Edge Devices

Lim, Cheol-Sun

Major in IT Convergence Enginnering

Dept. of IT Convergence Enginnering

The Graduate School

Hansung University

Many research works have been conducted to optimize computations of DNN(deep neural network) models in edge devices. Especially in autonomous driving, due to network latency and security issues, executing DNN models inside embedded systems is preferred rather than cloud servers. In case of autonomous driving, the number of DNN models has been continuously increased to process many raw sensor data. In this paper, to optimize the computational behavior of DNN models, we study DNN models in two different ways, and then, we propose each optimization technique. First, we apply NAS(Neural Architecture Search) approach to obtain optimized DNN models for edge devices. At here, we focus on a multi-objective reward function of MnasNet providing optimization tools in terms of the accuracy and the execution time. The

goal of MnasNet is to explore DNN models with the highest accuracy while satisfying the execution time constraints. Based on this, we propose a new reward function to explore DNN models with the maximum accuracy while ensuring that the inference is completed within the time limit. Second, we propose CPU-GPU Co-Scheduling framework that orchestrates the CPU as well as the GPU for multi-DNN execution. In the development of the proposed framework, we conducted further analysis for definitely finding the overhead that inevitably incurred when multiple DNNs are executed in a multiple-context environment and a single-context environment. Moreover, our framework utilizes unused CPU cycles for DNN computations to ease the computational burden of the GPU. For seamless communication between the CPU and the GPU, we propose low overhead data synchronization method. Finally, we apply layer execution time prediction model to select the appropriate core whenever a new DNN layer execution is issued. After applying the proposed solutions, the performance of multi-DNN execution is improved and the execution time is reduced by up to 46.6% compared to the GPU-only solution.

【KEYWORD】multiple-DNNs, NAS, GPU computing, parallel computing, heterogeneous computing