이종 하드웨어 가속기를 포함하는 모바일 플랫폼을 위한 시스템 수준의 딥 러닝 추론 최적화 기법

SYSTEM-LEVEL OPTIMIZATION FOR DEEP NEURAL NETWORK INFERENCE ON HETEROGENEOUS MOBILE PLATFORM

하순회 & 강두석 & 오진우 & 최종우

November 7, 2018

CONTENTS

LIST OF FIGURES

1 과제에 대하여

1.1 과제의 목적

본 과제의 가장 핵심적인 목적은 최근 중요성이 증가하고 있는 edge computing을 수행하기 위해서 embedded device 상에서 CNN 추론의 효율을 높히는 것이다. 이는 최근 인기를 끌고 있는 많은 모바일용 CNN 라이브러리들이 목적하는 embedded Device 상에서 CNN 추론을 수행하는 것에서 한발 더 나아간 것이라고 할 수 있다.

CNN 추론의 효율을 높히기 위한 많은 기존의 연구들은 data- parallelism을 이용하여 layer의 수행시간을 줄이는데 집중한 경향이 있다. 하지만 data-parallelism만으로 수행 효율을 높히는데는 한계가 있기 때문에, 레이어들을 하나의 task로 보고 task-parallelism을 이용한 스케줄을 통해 CNN 추론의 효율을 높히기로 하였다. 이에 따라 본 과제의 수행방식을 특정한 스케줄을 제공하기보다는 스케줄을 위한 알고리즘을 개발하는 것으로 결정하였다. 알고리즘 개발 이후, 해당 알고리즘이 기존 단일 PE 전담 방식 CNN 추론보다 효율적임을 보이며, 다양한 상황에서 적용이 가능함을 검증한 결과를 보이는 것을 최종적인 목적으로 삼았다.

1.2 선행 연구 조사

모바일 디바이스 상에서 CNN을 수행시키기 위한 라이브러리들은 다수 존재하고, CNN의 Layer 수행 효율을 올리는 연구는 kernel merging 등의 연구 결과가 있으나, CNN의 layer들을 별개의 task로 현재까지 발표된 논문 중에는 DeepX만이 본 과제와 유사한 연구를 진행한 것으로 파악되고 있다.

기존에 발표된 라이브러리들은 일반적으로 모바일 device 상에서 CNN을 수행하는 것에만 집중한 나머지, 여러가지 효율에 대해서는 고려되지 않은 채, CPU, GPU 혹은 CNN을 위해 특별히 고안된 NPU에 CNN 추론을 전담시키고 나머지 PE들은 CNN 추론이 완료되는 것을 기다리는 방식으로 CNN 추론을 수행하도록 만들어져 있었다. 하지만 삼성 갤럭시 S9을 비롯한 최신의 모바일 device들은 강력한 Hetrogeneous computing environment를 보유하고 있다: 다시 말해서 하나의 PE에 CNN을 전담시키고 나머지 PE들이 유휴자원으로 쉬고 있는 수행 방식은 시간이라는 측면에서 비효율적이라고 할 수 있다.

DeepX는 이러한 비효율을 전력 소모라는 측면에서 개선하였다. 이는 DeepX가 목표한 CNN 추론의 사용처는 장시간 지속적으로 추론이 요구되는 어플리케이션들이었기 때문이다. 하지만 본 과제에서는 반대로 짧은 시간동안 집중적으로 CNN을 수행하는 어플리케이션들(사용자 인식, 음성 명령 인식 등)을 목표로 삼아 정해진 시간 내에 CNN 추론의 처리량을 최대화 하는 것을 목적 삼았다.

1.3 과제 진행을 위한 인원 구성

본 과제는 크게 스케줄링 팀과 검증 팀으로 구성되어 과제를 수행한다. 스케줄링 팀은 여러 종류의 CNN configuration에 대한 파싱을 기반으로 한 개의 딥 러닝 네트워크를 가장 효율적으로 스케줄링하는 방법을 고안한다. 이후 검증 팀은 스케줄링 팀에서 만든 스케줄러가 최종적으로 도출해낸 스케줄링을 이종 하드웨어 가속기로 구성된 모바일 플랫폼에서 실제로 수행하여 스케줄링의 타당성을 검증한다.

본 문서는 스케줄링 팀이 구현한 스케줄러에 대한 자세한 개발 내용 설명과 스케줄링을 검증하는데 있어 발생한 이슈에 대한 설명을 목적으로 한다.

1.4 과제 수행 계획

중간 보고 일인 11/8일 현재 연구 환경 조성, 선행 연구 조사는 마무리가 되었다. 스케줄링 팀은 GA와 ILP로 파트가 나뉘어 있으며, 스케줄링 연구가 일단락 되어 있다. 검증 팀은 현재 스케줄링

단계	내용	기간(18/04/01 ~ 19/03/31)												비고
		4	5	6	7	8	9	10	11	12	1	2	3	
1	연구 환경 조성													
2	선행 연구 조사													
3	~~커널 최적화 연구 및 개발~~													~~결과물 공유~~
4	~~예측 모델 연구 및 개발~~													~~결과물 공유~~
5	Mapping 및 Scheduling 연구 개발													결과물 공유
6	Mapping 및 Scheduling 검증 및 개선													
7	최종보고서 제출													

Figure 1: 과제 수행 일정

팀이 넘겨준 스케줄을 검증하기 위해서 ARM Compute Library(이후 ACL)을 이용한 실험 환경을 조성하였으며, thread 간의 deadlock 문제를 해결하고 있다.

2 BACK GROUND

2.1 과제의 진행을 위한 고려 사항

본 과제의 수행 위해 가장 먼저 고려한 사항은 하드웨어이다. 일반적인 PC 환경에 비해서 다양한 제약 조건이 있는 임베디드 환경에서는 고성능 컴퓨터와 비교하여 상대적으로 느린 속도로 응용이 수행될 수 밖에 없다. 하지만 하드웨어의 특징과 장점을 최대한 고려한 스케줄링을 통하여 CNN 추론을 최대한 효율적으로 사용할 수 있을 것으로 보았다.

두번째 고려된 사항은 범용성이다. 하드웨어를 고려한 스케줄에 천착한 나머지, 특정 하드웨어에서 특정한 CNN만을 사용한 스케줄을 제공한다면, 이는 하루가 다르게 새로운 모바일 device가 출시되고, 매달 새로운 CNN이 발표되는 환경에서 금방 obsoleted 될 수 밖에 없다. 따라서 기본적으로 다양한 network에 적용이 가능하고, 가능한 하드웨어의 변화에 대응이 가능한 연구가 되어야 했다.

2.2 CNN 스케줄링을 위한 기본 정책

CNN 네트워크는 레이어마다 다른 메모리를 사용하며, 각 레이어별로 data의 의존성은 있지만 computation에는 의존성이 없기 때문에 각각의 레이어를 서로 다른 task로 볼 수 있다. 따라서 CNN 네트워크는 Task graph로 표현될 수 있다. 또한 이렇게 분리된 task들 간에 존재하 data 의존성은 더블 버퍼링을 통해서 data를 임시로 보관하여 없앨 수 있다. 이렇게 layer 간의 computational dependancy와 data dependancy를 제거하여 달성한 task-parallelism을 레이어 간 병렬화(Inter-layer Parallelism)이라고 하며, 본 과제에서 단위 시간당 처리량을 최대화하는 스케줄링의 최적화의 주된 수단로 삼았다.

추가적으로, 사측의 요청에 따라 CPU 이용률(utilization)에 제약 사항이 존재할 때, 이를 고려한 스케줄링 또한 개발하였다.

본 연구에서 스케줄링의 범용성을 보이기 위한 네트워크로 SqueezeNet, DenseNet(k=32), MobileNet v1, MobileNet v2를 사용하였다. 이는 각 CNN들의 특징을 고려하였다. Mobilenet v1은 모바일 디바이스를 위한 CNN이라는 이름처럼, 간결한 sequential 구조를 가지고 있기 때문에, 기본적인 스케줄의 유효성을 보이기 위해 선택되었다. Mobilenet v2는 Mobilenet v1와 비슷하지만 더 많은 레이어를 지녀서 레이어 갯수에 따른 복잡도 변화에 대응할 수 있을지 보이기 위하여 선태되었다. SqueezeNet은 Sequential structure가 아닌 parallel structure로 구성되었기 때문에, parallel 스케줄링을 확인하기 위해서 선택되었다. DenseNet(k=32)은 Dense Block이라는, 여러 concatenate layer들로 이루어진 특별한 레이어로 구성되어 있다. 따라서 이런 레이어 블럭들을 적절히 스케줄링 되는지 보이기 위하여 선택하였다. 상기한 4개 네트워크들 대상으로, 본 문서에서는 우리가 개발한 스케줄러의 전체적인 구조와 동작(섹션2, 3, 4), 스케줄러 각 부분의 설계(섹션5), 각 부분의 동작(섹션6), 마지막으로 스케줄러 검증에서 발생한 이슈에 대하여 자세하게 설명한다. 이번 프로젝트를 통해 구현한 스케줄러를 편의를 위해 본 문서에서는 가칭 "Hetero-스케줄러"라고 명명한다.

3 SW 동작

Figure 1은 Hetero-스케줄러의 전체적인 구조를 보여준다. 우선 Hetero-스케줄러는 두 가지 종류의 input을 요구한다. 첫째는 네트워크 configuration 정보이다. Hetero-스케줄러는 앞서 언급한 네트워크 중 하나를 선택하는데, 이 때 input 포맷에 맞게 해당 네트워크 configuration을 파싱한다. 본 과제에서 제시하는 스케줄러는 caffe prototxt를 input 포맷으로 사용하는데, 이에 대한 자세한 내용은 섹션 3.1에서 설명한다.

둘째는 네트워크 레이어에 대한 PE(Processing Element)별 연산 수행 시간 정보다. 이 또한 첫 번째 input처럼 caffe prototxt 포맷을 사용한다. 보다 정확한 스케줄링을 위하여 레이어 프로파일링은 타겟 하드웨어(Galaxy S9)에서 직접 수행했다. 그리고 일부 프로파일링은 CPU frequency 기반의 interpolation으로 처리했는데, 이에 대한 자세한 내용은 섹션3.1에서 설명한다.

이러한 input 정보를 기반으로 Hetero-스케줄러는 task graph 자료 구조를 생성한다. Task graph는 network inference 상의 의존성 정보, 각 레이어의 크기, 각 레이어가 처리하는 데이터의 크기 등의 정보를 담는데, 이를 이용하여 스케줄링을 처리한다. Task graph 자료구조에 대한 자세한 내용은 섹션 5.1에서 설명한다.

Scheduling을 위한 위의 전처리 과정이 모두 끝나면, Hetero-스케줄러는 본격적인 스케줄링에 돌입한다. 본 과제에서 구현한 스케줄러는 2가지로, GA 스케줄러와 ILP 스케줄러가 이에 해당한다. GA 스케줄러는 GA 알고리즘을 이용하여 효율적인 스케줄링을 돌려준다. GA 알고리즘은 매우 방대한 문제 공간에서의 최적해를 찾는 데 쓰일 수 있는데, 본 과제가 목표로 삼는 효율적인 스케줄링 탐색이 이와 상응한다고 볼 수 있다. 네트워크가 담고 있는 많은 종류의 레이어에 대해 이종 연산 장치 단위의 좋은 스케줄링을 찾는 문제이기 때문이다. 한편 ILP(Integer Linear Programming) 스케줄러는 여러 선형 수식들을 산정하여 문제를 해결하기 때문에, 수식이 잘 정의되어 있다면 최적(optimal) 스케줄링을 도출할 수 있다. 유전 특성을 이용하는 GA 스케줄러는 이에 비해 보다 덜 최적의 스케줄링을 도출할 가능성이 크지만 ILP 스케줄러보다 더 복잡한 케이스의 스케줄링을 가능케 한다는 장점을 지닌다. 상기 두 가지 스케줄러에 대한 자세한 내용은 섹션4에서 설명한다.

참고로 본 과제는 Python 언어를 사용하여 스케줄러를 구현하였다. 하지만 ILP 스케줄러는 Gurobi라는 C계열 기반의 플랫폼을 사용하기 때문에 이를 Python 언어와 연동하기 위해서 동적 라이브러리를 생성하여 포팅한다. 이에 대한 자세한 내용은 섹션3.3에서 설명한다.

스케줄러에 의해 산출된 스케줄링 결과는 각 레이어의 수행 시간과 mapping 정보가 담긴 간트 차트(Gantt Chrat)로 표현된다 (Figure 2). 이는 스케줄링에 대한 하드웨어 상의 실제 검증 단계에서 비교 대상으로 사용된다. 검증 관련 이슈에 대한 설명은 섹션7에서 설명한다.

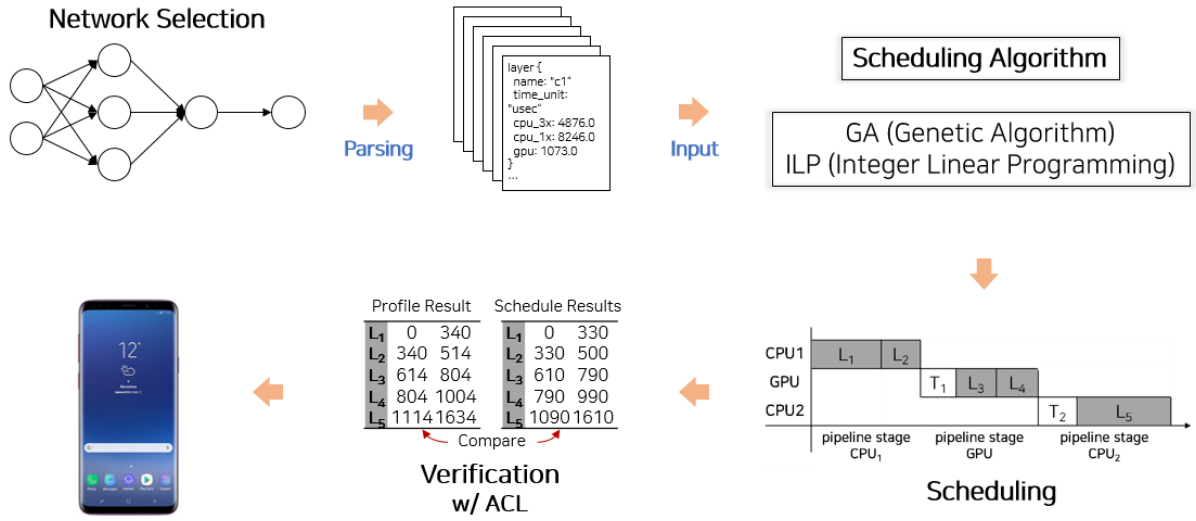

Figure 2: Hetero-스케줄러의 전체적인 진행 과정

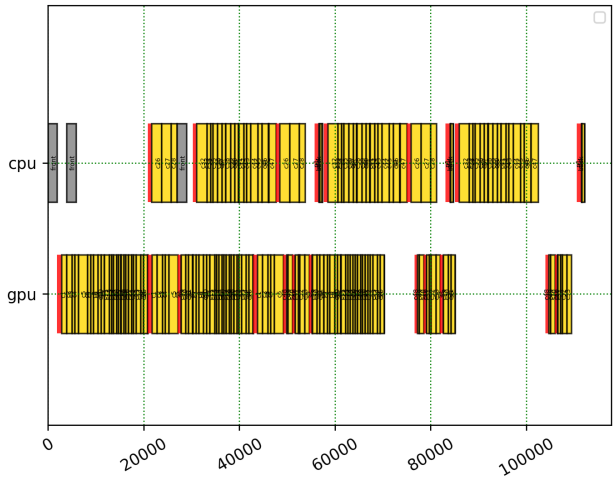

Figure 3: 두 개의 Image에 대한 스케줄링 결과 - 간트 차트(Gantt Chart)

4 SCHEDULER를 위한 전처리

4.1 Input parsing

앞서 언급했듯이, Hetero-스케줄러는 네트워크 configuration 정보와 네트워크 레이어에 대한 PE별 연산 수행 시간 정보를 입력으로 받는다. 그리고 이들 input들 모두 caffe prototxt 포맷을 차용한다는 것 또한 설명하였다. Prototxt는 유명 딥 러닝 플랫폼인 Caffe에서 사용하는 protocol buffer 형식의 파일이다. Prototxt는 효율적인 직렬화(Serialize)와 여러 언어를 위한 효율적인 인터페이스 구성을 장점으로 취한다.

Figure 3은 이러한 prototxt input 정보에 대한 실제 사용되는 코드의 일부다. 왼쪽은 네트워크 configuration 정보를 담고 있는 model prototxt, 오른쪽은 PE별 레이어 수행 시간 정보를 담고 있

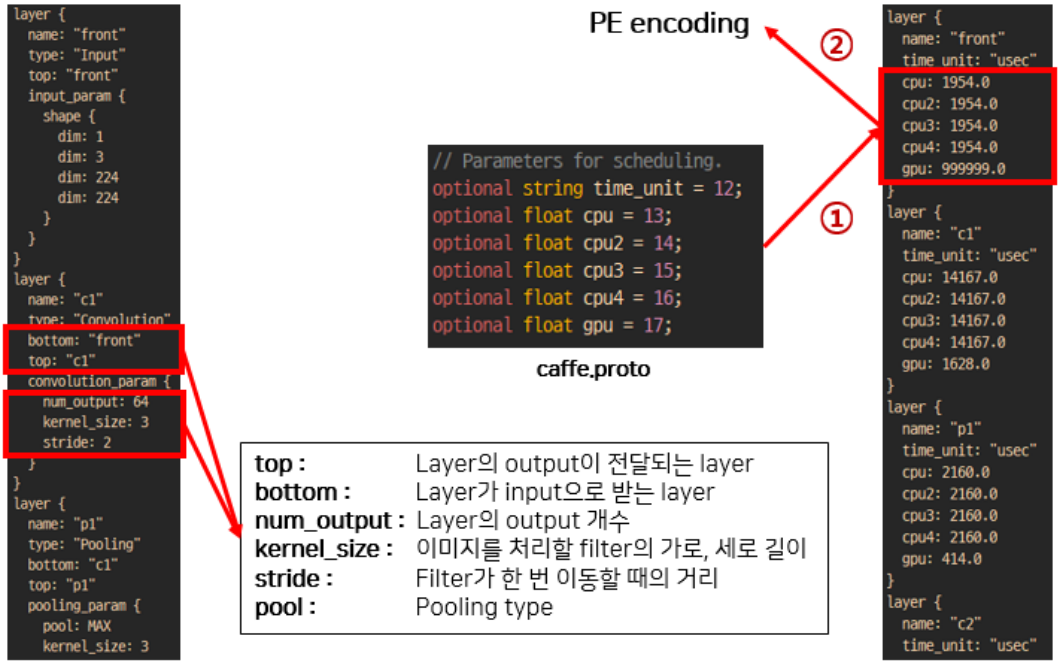

Figure 4: Hetero-스케줄러가 받는 입력 정보 - model prototxt, estimation prototxt

는 estimation prototxt이다. Prototxt 포맷의 파일을 R/W 하기 위해선 caffe.proto 파일이 필요한데, 가운데 상단에 보이는 것이 그것의 일부분이다. Caffe에서 기본적으로 제공되는 caffe.proto 파일은 LayerParameter라는 타입을 정의하고 이는 name, type 등의 attibute를 포함한다. PE(cpu, cpu2, cpu3, cpu4, gpu)에 대한 정보는 본래 담고 있지 않다. 그래서 본 과제는 PE별 레이어 스케줄링을 위해서 caffe.proto 파일에 그림과 같이 PE attribute들을 추가했다. 한편, 실제 스케줄링 시 각 PE는 정수로 encoding 된다. 즉 cpu cpu4는 정수 0 3에, gpu는 정수 4로 치환된다.

4.1.1 *Direct layer Profiling*

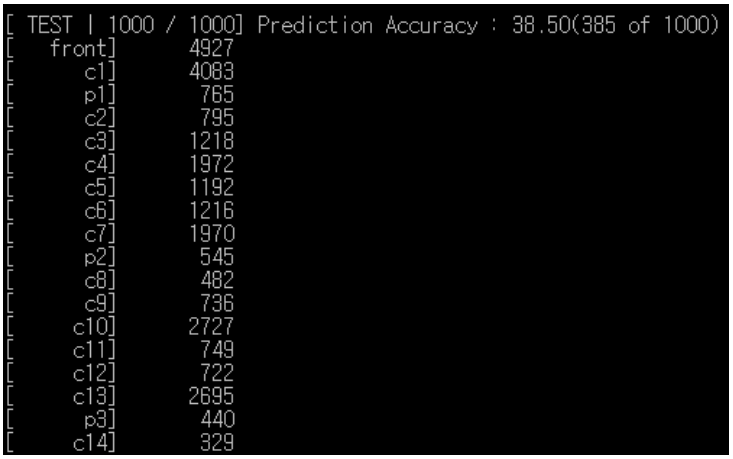

Figure 5: 타겟 디바이스(Galaxy S9) 상에서의 레이어 프로파일링

Estimation prototxt를 완성하기 위해서는 네트워크 레이어별 수행 시간 프로파일링 결과가 필요하다. 이종 하드웨어 가속기를 포함하는 모바일 플랫폼을 위한 스케줄링 최적화가 본 과제의 목표이기 때문에, 이를 위한 정확한 레이어 프로파일링이 필요하다. Figure 4는 ADB(Android Debug Bridge)

shell로 타겟 디바이스(Galaxy S9)에 접속하여 실제 레이어 프로파일링을 한 모습이다. 타당한 프로파일링을 위해 네트워크 inference의 정확도(Accuracy)를 항상 검사한다.

4.1.2 *CPU frequency-based interpolation*

앞서 말했듯이, 본 과제에서는 Intra-layer 병렬화 뿐만 아니라 Inter-layer 병렬화도 고려한다. 추가적으로, 본 과제에서는 하나의 CPU core에 대해서도 하나의 PE로 보아 더 나은 효율의 스케줄링을 도출하는 스케줄러를 구상 및 구현하였다. 이에 따라, Figure 5에서 보듯 다양한 PE 분배에 대한 스케줄링을 관찰할 수 있다.

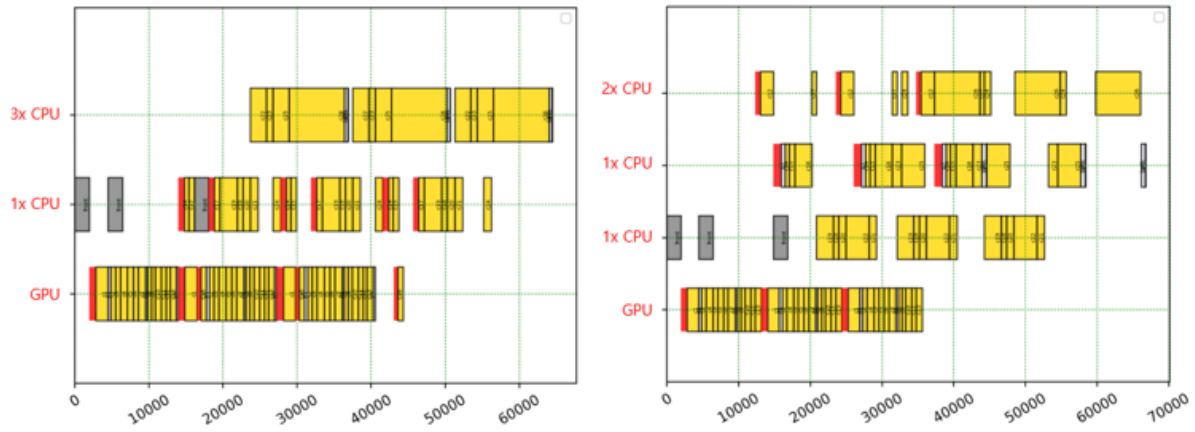

Figure 6: 하나의 CPU 코어의 PE 적용에 따른 다양한 스케줄링

한편, 타겟 하드웨어에서 사용하는 ACL(Arm Compute Library)은 내부의 거버너(Governer)라는 요소가 online 된 CPU의 수에 따라 주파수를 바꾸어버리는 특징을 보인다. 즉, 이는 실제 inference 시 적용되는 Fully-utilized frequency(1.79GHz)가 아닌 다른 값의 frequency 상에서 프로파일링이 이루어진다는 의미다. 예를 들어, Figure 5의 3x CPU PE에 대한 프로파일링은 2.31GHz frequency로 이루어진다. 따라서, Fully-utilized frequency(1.79GHz)에 대한 PE의 프로파일링 frequency(2.31GHz)의 비율(2.31/1.79)만큼 선nterpolation하여 프로파일링 결과를 보정한다.

4.2 ILP porting

섹션2에서, Python 언어로 구현된 GA 스케줄러와는 달리 ILP 스케줄러는 C 계열 기반의 언어로 이루어져있기 때문에 이를 포팅한다고 언급하였다. Python 패키지 중 ctypes 패키지는 이를 가능케 한다. 먼저 ILP 스케줄러의 핵심적인 함수들을 공리 라이브러리(*so, shared library)로 래핑하고 ctypes를 이용하여 공유 라이브러리를 로드한 후, 이를 Python 모듈 안에서 실행하는 구조를 지닌다.

```
self.porting = CDLL('./main_porting.so')
def do_schedule(self):
    self.porting.ilp('mobilenet_v1_cfg_to_prototxt.prototxt', 'mobilenet_v1_estimation_cpu_core_4.prototxt', self.objectives)
```

Figure 7: GA 스케줄러와의 연동을 위한 ILP 스케줄러 포팅

5 SCHEDULER 진행 흐름

5.1 GA Scheduler

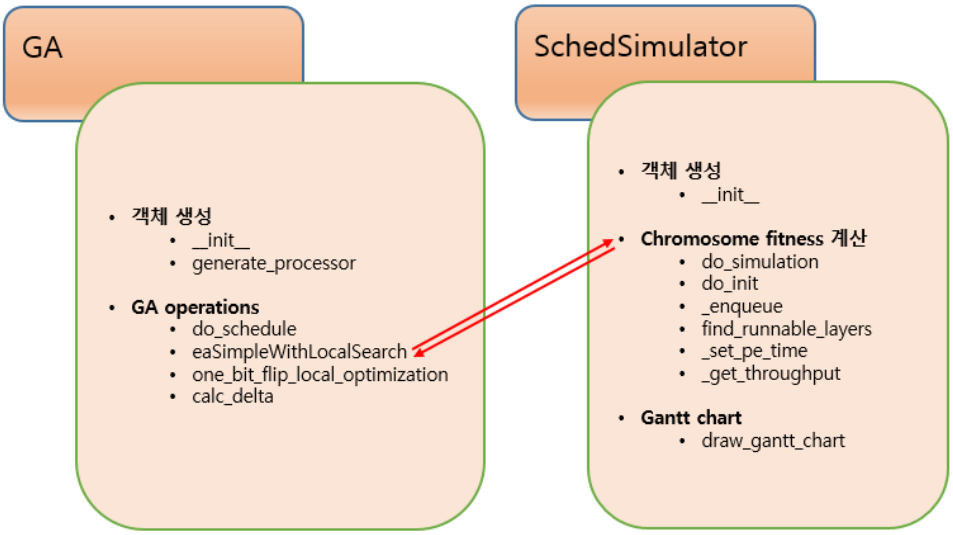

Figure 8: GA 스케줄러의 진행 흐름

본 섹션에서는 Figure 7에 정리된 GA 스케줄러의 전체적인 진행 과정을 설명한다. GA 스케줄러는 크게 GA 클래스와 SchedSimulator 클래스로 구성된다. GA 클래스는 GA 객체를 생성하고 GA operation 관련 함수들을 호출할 때 이용되며, SchedSimulator 클래스는 SchedSimulator 객체를 생성하고 GA chromosome의 fitness 값 계산, 스케줄링 결과를 간트 차트로 변환하는 역할을 한다.

5.1.1 *GA part*

__INIT__ GA 스케줄러의 GA 연산을 위해서 초기 객체를 생성한다. 모집단(population)의 크기를 비롯하여, Fitness 값의 방향(최소/최대), 자손들(individual, offspring)의 생성 방식과 fitness 계산 방식, GA 연산(selection - crossover - mutation - replacement)에 쓰일 함수 등을 지정하는 역할을 한다. 본 과제의 GA 스케줄러는 각 GA 연산에 대하여 다음과 같은 알고리즘을 적용한다.

1. Selection: Tournament selection
2. Crossover: Uniform crossover
3. Mutation: Multiple Flip bit
4. Replacement: Updating all of population

GENERATE_PROCESSOR 위의 __init__ 함수에서 자손들의 생성 방식에 적용되는 함수다. 스케줄링이 몇 개의 PE로 이루어지는지에 따라서 generate_processor 함수가 생성하는 chromosome 구성이 약간씩 다르다. 예를 들어, PE의 개수가 5개라면 chromosome 안의 유전자(gene)가 가질 수 있는 값은 0 4 사이의 정수값이다.

DO_SCHEDULE

EASIMPLEWITHLOCALSEARCH

ONE_BIT_FLIP_LOCAL_OPTIMIZATION

CALC_DELTA

5.1.2 *SchedSimulator*

__INIT__ GA 스케줄러의 fitness 값, 즉 1/throughput 값을 측정하고 스케줄링을 위한 간트 차트를 작성하기 위한 초기 객체를 생성한다.

DO_SIMULATION

DO_INIT

_ENQUEUE

FIND_RUNNABLE_LAYERS

_SET_PE_TIME

_GET_THROUGHPUT

DRAW_GANTT_CHART memory transition time에 대한 구체적인 설명 및 계산법 설명.

5.2 ILP Scheduler

6 SCHEDULER 구조 설계

6.1 GA Scheduler

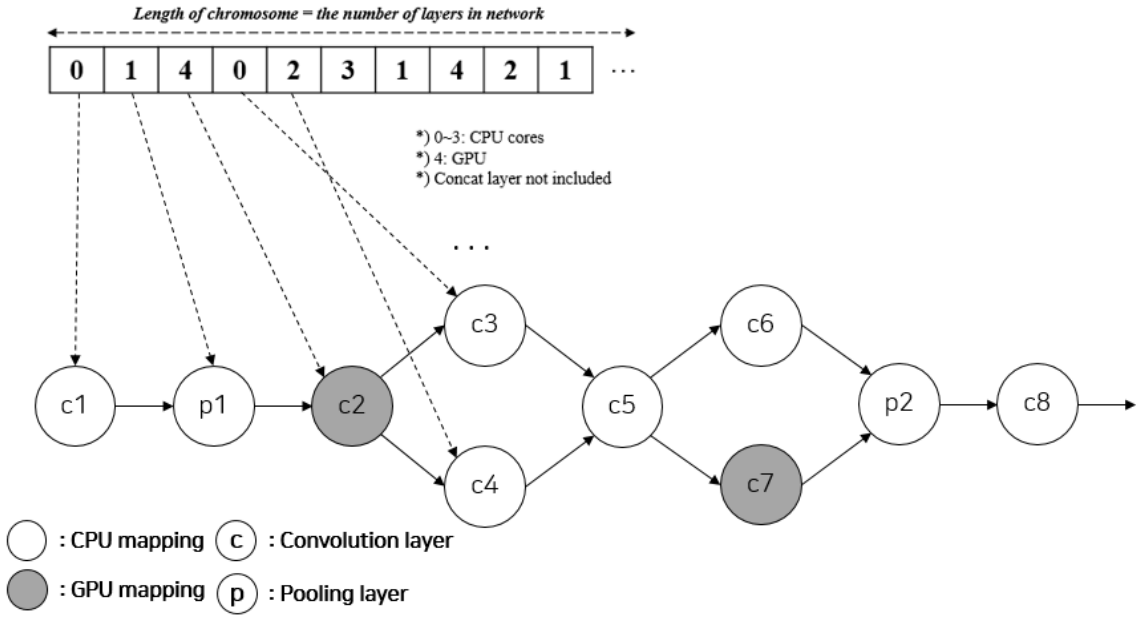

Figure 9: Chromosome 구성

4개의 코어로 이루어진 CPU가 있다고 가정할 때, 코어 1개마다 PE로 설정할 수 있다. 그리고 GPU 프로세서 또한 하나의 PE로 볼 수 있으므로, 총 5개의 PE 구성을 이룰 수 있다. 따라서 Input으로 들어오는 model prototxt에도 이와 같은 PE 단위로 정수 0 4로 인코딩된다. GA 스케줄러의 chromosome은 하나의 유전자(gene) 정보를 하나의 PE(정수)로 설정한다.

generate_processor 함수를 통해 이러한 유전자를 설정하는 방식은 여러 가지이다. 하나는 무작위로 값을 배치하는 방법이고, 다른 하나는 chromosome 안의 서로 인접한 유전자들이 부분적으로 같은 값을 가지도록, 즉 clustering을 이루게끔 유도하는 방식이다. Clustering의 목적은 chromosome(mapping)의 throughput을 높여 GA fitness 값을 최적에 가깝게 하기 위함이다.

task graph 설명 추가 GA model flow 그림 추가 GA chromosome 그림 추가

6.2 ILP Scheduler

ILP 스케줄러의 전체적인 진행 과정을 중점적인 함수들에 대한 기술로 추상화하여 설명.

7 SCHEDULER 동작 설계 & 구현

7.1 GA Scheduler

throughput estimation 계산법 설명(그림 및 수식 첨부)

7.2 ILP Scheduler

ILP란 수식을 통해서 최적해를 찾는 최적화 기법으로, 여러 부식으로 표현된 시스템 모델에 제약식을 추가하여 원하는 조건을 설정한다. 이렇게 설정된 조건 아래서 구하고자 하는 바를 목적식으로 표현하여 최적해를 구한다.

7.2.1 *ILP*에서 사용하고 있는 가정들

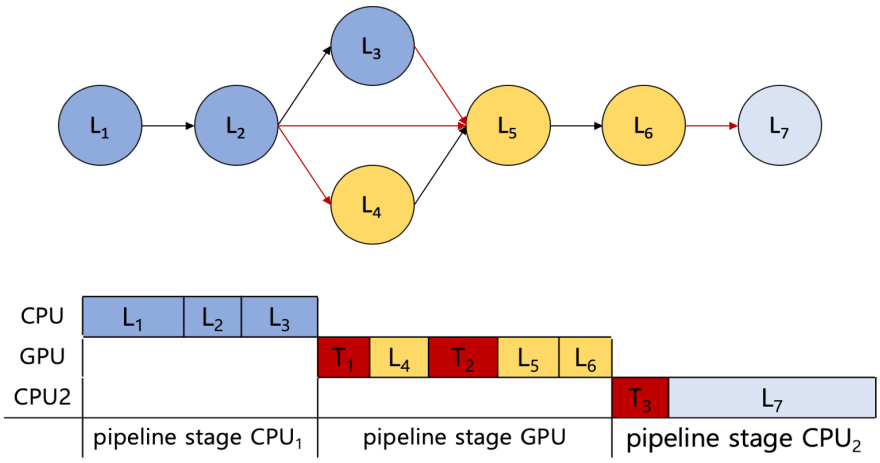

Figure 10: ILP의 진행 흐름

ILP의 진행 흐름에서 볼 수 있듯이, CNN 상의 레이어들을 각각의 mapping 가능한 task로 보았다. 즉, layer 단위로 매핑을 진행하였으며, Task parallelism 중에서 pipelined schedule을 사용

하였다. 자유로운 Task parallelism을 선택하지 못한 이유는, 1/throughput을 ILP 상에서 계산하기 어려웠기 때문이다. pipelined schedule을 시스템 모델로 설계함으로써 최대 pipeline stage를 1/throughput으로 계산할 수 있게 되었다.

또한 관찰을 통해서 각 PE 상에서 하나의 layer(task)를 처리하는 시간과 Memory transition time은 input으로 받아서 수식의 기본 상수로 삼았다.

7.2.2 *ILP*의 목적식

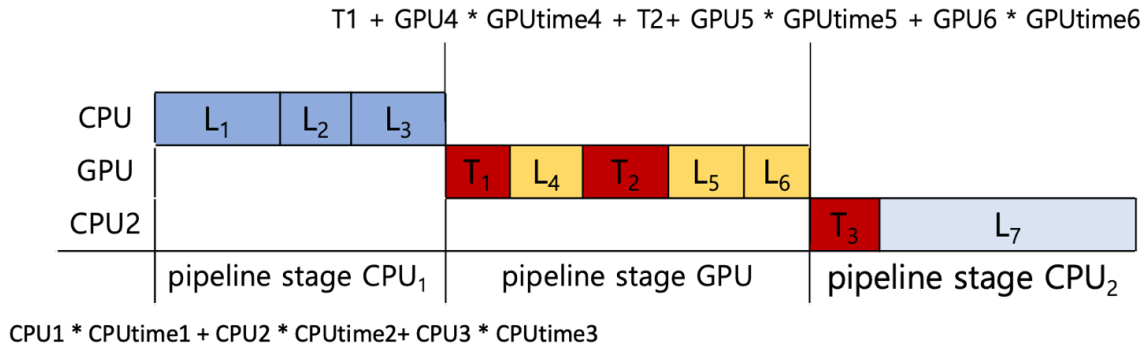

Figure 11: ILP의 목적식

시스템 모델에서 memory transition의 크기는 보내는 쪽의 정보를 따르고, 발생 시간은 받는 쪽에서 요청할 때로 가정하였다. 따라서 파이프라인 스테이지는 한 PE가 처리하는 모든 레이어들의 합과 memory transition에 의한 overhead의 합으로 계산할 수 있다.

따라서 현재 ILP의 목적식은 다음과 같다.

1. minimizing(Max.pipline_stage) = minimizing(Sum.PE_computation + overhead)

7.2.3 *ILP*에서 *input*을 통해 미리 받는 상수

ILP 수식을 세우기 위해서 일부 정보는 input을 통해서 미리 받는다.

1. CPUtimej[i]
   i번째 레이어를 j번째 CPU에서 처리하는데 걸린 시간.
2. GPUtime[i]
   i번째 레이어를 GPU에서 처리하는데 걸린 시간.
3. transtimeGC[i]
   GPU에 매핑된 i번째 레이어에서 메모리 transition이 발생할 때 걸린 시간. 해당 상수는 메모리 크기와 메모리 통신시간의 regression 식을 통해서 도출된다.
4. transtimeCG[i]
   CPU에 매핑된 i번째 레이어에서 메모리 transition이 발생할 때 걸린 시간. 해당 상수는 메모리 크기와 메모리 통신시간의 regression 식을 통해서 도출된다.

7.2.4 *ILP*에서 사용된 변수들

가장 먼저 layer의 시간정보를 저장할 변수들을 설정하였다.

1. layerstart[i]
   i번째 레이어의 시작시간을 표현하는 실수 변수. 해당 변수들 중에서 강제로 설정하는 변수는 layer_start[0]으로, 0번째 레이어의 시작시간은 언제나 0이다.
2. layerend[i]

i번째 레이어의 종료시간을 표현하는 실수 변수. if (i = 마지막 레이어)일 때 layer_end[last]는 전체 스케줄의 latency와 같다.

이후, 시스템 모델을 표현하거나 다른 제약식을 위해서 사용된 binary 변수들을 설정하였다.

3. CPU(each)[i]
   i번째 레이어가 어떤 CPU에 매핑되었는지 여부를 밝히는 binary 변수이다.
4. GPU[i]
   i번째 레이어가 GPU에 매핑되었는지 여부를 밝히는 binary 변수이다. kernel merging을 고려하지 않았기 때문에, GPU는 한번에 한가지 작업만 가능한 큐 방식으로 동작한다. 따라서 GPU는 여러개의 PE로 분할하지 못한다.

i번째 레이어의 dependancy와 직전 레이어와의 관계를 표현하는 binary 변수들은 다음과 같다.

5. preCPU(each)[i]
   i번째 레이어가 직접적인 의존성을 가진 레이어중 하나는 어떤 CPU에 매핑되어 있다.
6. preGPU[i]
   i번째 레이어가 직접적인 의존성을 가진 레이어중 하나는 GPU에 매핑되어 있다.
7. CPU(each)pipelinestart[i]
   i번째 레이어는 어떤 CPU의 pipeline 스테이지가 시작하는 레이어다.
8. GPUpipelinestart[i]
   i번째 레이어는 GPU의 pipeline 스테이지가 시작하는 레이어다.
9. same[i][j]
   i번째 레이어와 j번째 레이어는 같은 PE에 매핑되었다.
10. transCG(each)[i][j]
    i번째 레이어와 j번째 레이어는 직접적인 연결이 되어 있으면서 CPU->GPU memory transition이 발생했다. CPU를 여러개의 PE로 볼 때 map memory transition은 각각 발생한 것으로 보았다.
11. transGC[i][j]
    i번째 레이어와 j번째 레이어는 직접적인 연결이 되어 있으면서 둘 사이에 GPU->CPU memory transition이 발생했다.

7.2.5 *ILP*의 제약식

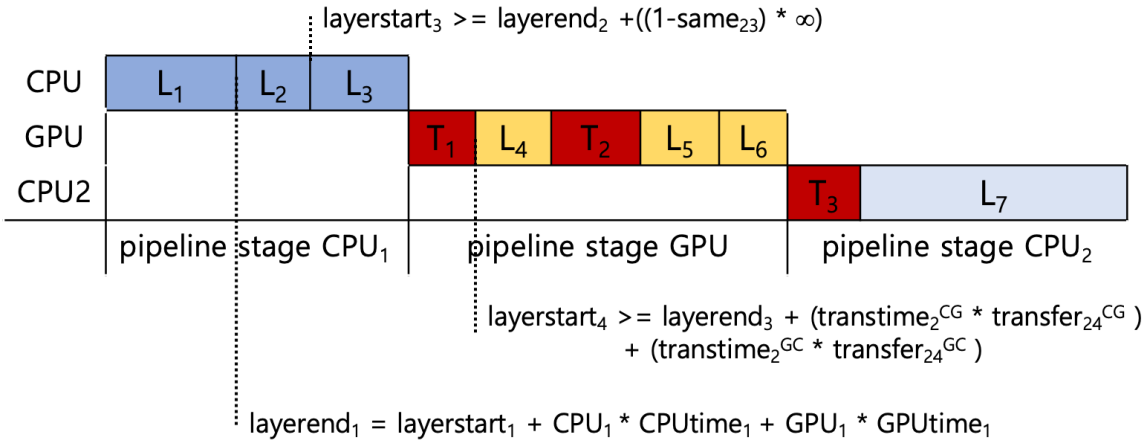

Figure 12: 제약식

1. layerstart[i] >= layerend[j] + transtimeGC[j] * transGC[j][i] + transtimeCG[j] * transCG[j][i]

첫번째로 설명할 제약식은 하나의 레이어는 자신의 선행레이어가 종료되고 만일 선행 레이어와 다른 PE에서 computation이 발생할 경우, transition시간이 끝난 이후에 시작이 된다는 것을 묘사한 수식이다.

2. layerend[i] = layerstart[i] + CPU[i] * CPUtime[i] + GPU[i] * GPUtime[i]

그림에 표현된 식 중 두번째로 설명할 식은 어떤 레이어의 종료 시간은 레이어의 시작 시간에 레이어에 매핑된 PE의 수행시간을 더하면 된다는 의미를 가진다.

3. layerstart[i] >= layerend[j] - ((1-same[j][i] * inf)

세번째로 설명할 식은 두개의 레이어에 병렬성이 있을 경우, 만일 두 레이어가 같은 PE를 사용한다면, layer 순서가 먼저인 j가 종료될 때까지 i번째 레이어는 시작이 될 수 없다는 뜻이다. 만일 두 레이어가 서로 다른 PE를 사용한다면, layerstart[i] >= -inf가 되므로 layer[i]는 다른 식으로 제약받지 않는 이상 아무데나 매핑이 가능하다는 뜻이 된다.

이후의 제약식들은 그림으로 표현되지 않은 식들이다.

4. CPU(each)[i] + GPU[i] = 1

그림에 표현되지 않은 제약식 중 가장 먼저 설명할 제약식은 하나의 레이어는 하나의 프로세서가 전담해서 처리한다는 가정을 표현하는 식이다. 각각의 레이어를 서로 다른 프로세스로 보면, 각각의 프로세스를 하나의 PE가 전담한다는 것은 각 프로세스가 독립적으로 동작하 현실에 부합한다고 볼 수 있다.

5. for (i = 0 -> size of layers)
   CPU(each)pipelinestart[i] + GPUpipelinestart[i] = Max.(3, Total number of PE)

두번째 제약식은 파이프라인 스테이지에 대한 식이다. 최초의 가정 중에 memory transition이 적을수록 overhead가 적을 것이므로, 전반적인 성능이 좋아질 것이라는 가정이 있었다. 이런 가정과 한 PE마다 다른 파이프라인 스테이지를 책임질 것이라는 가정을 합쳐서 다음의 식을 만들었다.

3 or Total number of PE인 이유는 하나의 CPU만을 사용한 스케줄의 경우, 실제로는 전체 CNN의 시작부분과 끝부분이 붙어서 하나의 파이프라인이 되지만, ILP 상에서는 둘을 다른 파이프라인 스테이지로 보았기 때문이다. 둘 이상의 CPU를 사용하는 경우에는 현재 시작과 끝을 다른 CPU가 담당하게 만들어 전체 파이프라인 스테이지의 시작점 숫자가 Total number of PE가 되었다.

6. for (i = 0 -> size of layers)
   for (j = 0 -> i-1)
   layerstart[i] >= layerend[j] - ((1 - same[j][i]) * inf)

이 제약식은 두개의 작업을 하나의 프로세서에서 동시에 처리할 수 없다는 제약을 표현했다. kernel merging을 이용하면 하나의 프로세서에서 여러개의 작업을 하나의 작업처럼 처리할 수 있으나 kernel merging을 사용하지 않을 것이라고 전제를 하였기 때문에, 하나의 프로세는 한번에 하나의 작업만을 처리한다. j를 모든 레이어가 아니라 i의 이전 레이어에 대해서만 처리한 이유는 j를 모든 레이어에 대해서 처리하면 자기자신의 종료 시간보다 시작시간이 커져야한다는 모순된 제약 사항이 생기기 때문이다.

이후로 설명할 제약식들 주로 프로세서의 변동을 표현하기 위해서 사용하는 식들이 많다. pre를 사용하는 변수들은 순서상 앞의 레이어가 아니고 의존성을 가진 레이어라는 점을 기억해야한다.

7. CPU(each)pipelinestart[i] = CPU(each)[i] & (1 - preCPU(each)[i])
8. GPUpipelinestart[i] = rGPU[i] & (1 - preCPU[i])

의존성을 지닌 모든 레이어가 자신과 다른 PE를 사용하면 파이프라인 스테이지의 시작으로 표시한다.

9. transGC(each)[i][j] = GPU[i] & CPU(each)[j]
10. transCG[i][j] = (1 - GPU[i]) & GPU[j]

자신이 의존하고 있는 레이어가 GPU에 매핑되어있고 자신이 어떤 CPU에 매핑되어 있으면 GPU->CPU memory transition이, 자신이 의존하고 있는 레이어가 GPU에 매핑 되어 있지 않고 자신이 GPU에 매핑되어 있으면 CPU->GPU memory transition 발생한 것을 표시한다.

8 SCHEDULING 검증 관련 이슈

8.0.1 *ACL*에 대한 설명

현재 검증을 위한 도구로 Arm사에서 오픈소스로 제공하고 있는 Compute Library를 이용하고 있다. TensorFlow lite, PyTorch 등 유명한 여러가지 ML 라이브러리 대신 ACL를 사용한 가장 중요한 이유는, 바로 openCL 기반의 자유로운 interlayer 맵핑능력이다. 전술한 바, 현존하는 대부분의 CNN 라이브러리들은 단일 PE가 추론을 전담하는 간단한 스케줄을 사용하기 때문에 본 과제에서 필요로 하는 layer 별 PE 선택이 불가능한 경우가 많았다. 이 때문에 OpenCL을 직접 사용하여 하위 레벨부터 상위 레벨까지의 interface 모두 구현한 ACL을 사용하는 것이 불가피했다. 하지만 현재 ACL 상의 스레스 스케줄러의 동작을 정확하게 제어하지 못하여서 2/2, 3/1, 2/1/1 CPU 분할 정책에 대한 디바이스 실험은 진행하지 못하고 있다.

9 스케줄링 실험 및 결과

지금까지 서술한 방법론들을 사용하여 스케줄링을 진행하였다. 결과는 1/처리량의 시간 단위로 표현되었으며 짧을수록 한번의 CNN 추론에 걸리는 시간이 적다는 뜻이다. 따라서 작을수록 효율적인 결과이다.

CNN의 GPU/CPU(멀티코어, 1스레드) 수행 시간

CNN 종류	GPU 수행 시간	CPU 수행 시간
squeezenet	24541	37291
densenet	49671.2	91427
mobilenet_v1	28539.3	58671
mobilenet_v2	41188.7	79854

Figure 13: CPU only/ GPU only 스케줄의 수행시간

Figure 13은 과제의 baseline이라고 할 수 있는 GPU only/ CPU only 스케줄의 수행시간이다. GPU only/ CPU only 스케줄에서는 parallelism이 고려되지 않기 때문에 수행시간과 1/처리량의 시간이 동일하다.

CPU 사용 수준에 따른 최대 처리량

CNN 종류	CPU100%	CPU 70%	CPU 60%	CPU 50%
squeezenet	11763.7	15118.6	15650.7	17583
densenet	29532.7	30805.7	33770.7	33732.1
mobilenet_v1	16154.8	18195.8	20109.8	22008.8
mobilenet_v2	20713.8	23734.7	24749.7	26459.7

Figure 14: CPU Utilization Level에 따른 ILP 스케줄링 최적 결과

Figure 14에서 볼 수 있는 CPU Utilization Level에 따른 최대 처리량을 보면 CPU를 50%만 사용하는 케이스에도 GPU Only에 비해서 71.6%, 67.9%, 77.1%, 64.2%의 처리 시간만을 필요로 한 것을 확인할 수 있다.

9.0.1 *GA* 결과 정리

9.0.2 *ILP* 결과 정리

Figure 15: ILP 스케줄링 결과 비교

일반적으로 1111의 다중 스레드 CPU 정책이 가장 좋은 결과를 보여주고 있다. 다음은 100% CPU Utilization, 50% CPU Utilization에서 ILP를 통해 얻은 optimal scheduling의 gantt chart이다.

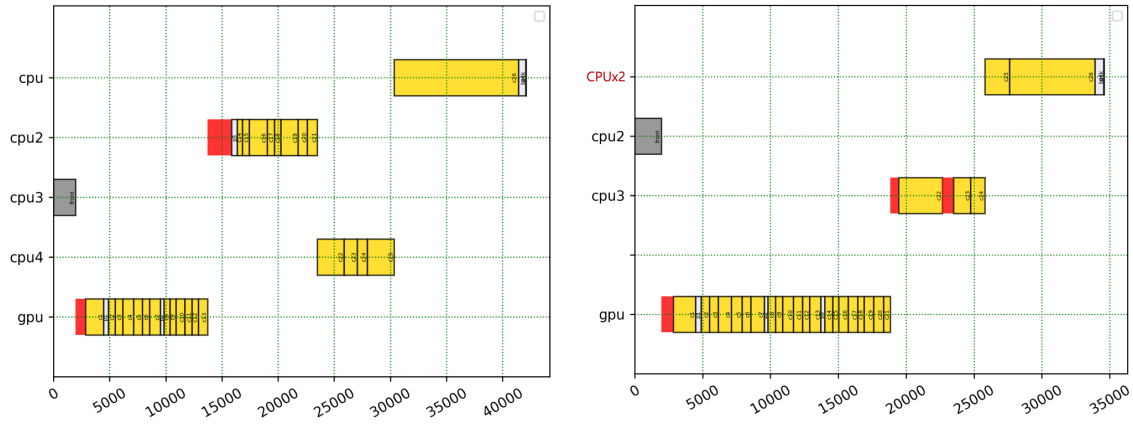

Figure 16: 좌: CPU Util 100%, 우: CPU Util 50%, Squeezenet

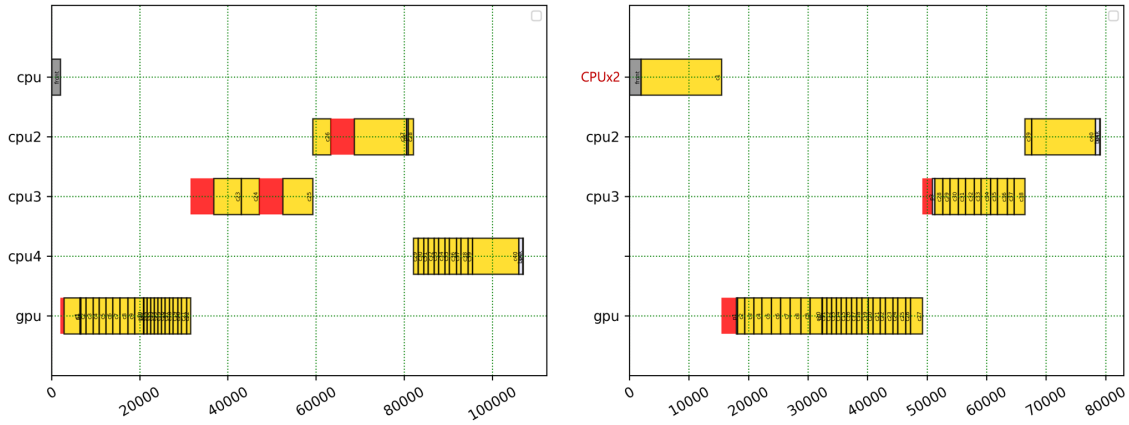

Figure 17: 좌: CPU Util 100%, 우: CPU Util 50%, Denseenet

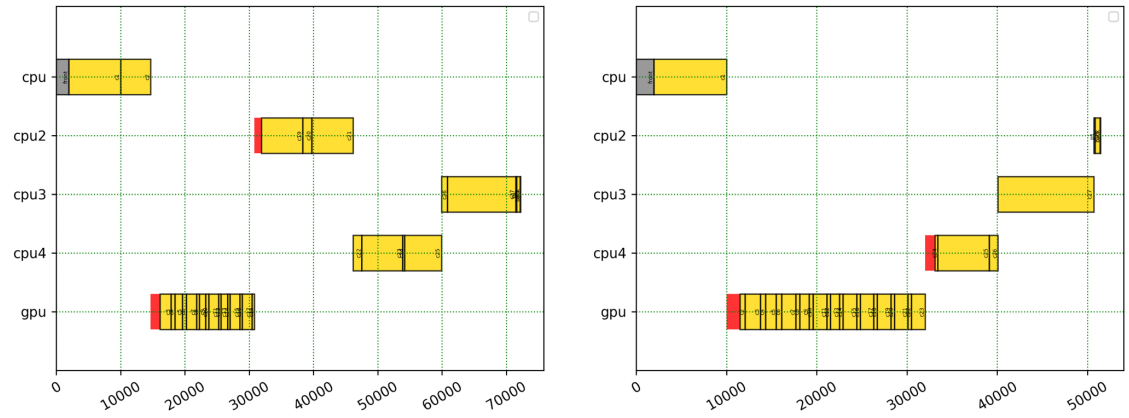

Figure 18: 좌: CPU Util 100%, 우: CPU Util 50%, Mobilenet v1

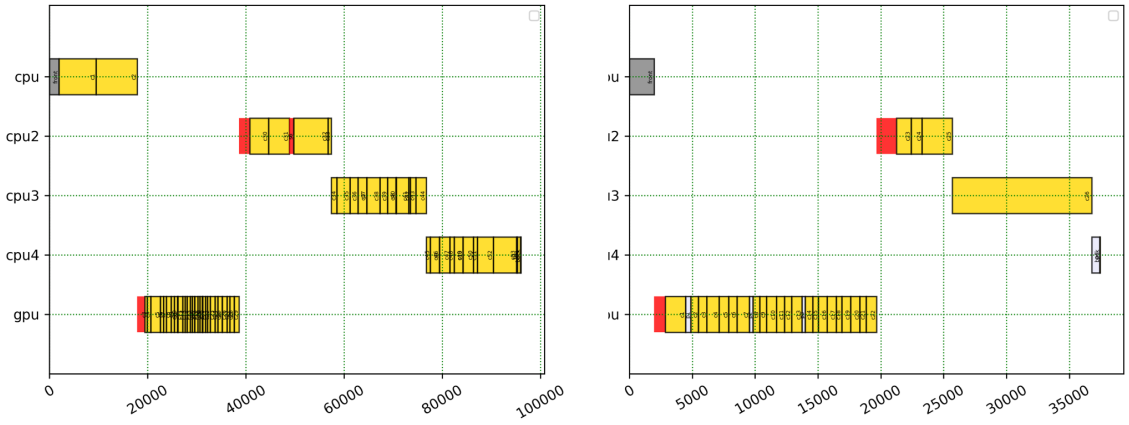

Figure 19: 좌: CPU Util 100%, 우: CPU Util 50%, Mobilenet v2

REFERENCES