단일 ISA 이기종 멀티 코어 구조를 위한 프로파일 기반 ArmCL 최적 스케줄 탐색

*차주형, **이주빈, ***권용인
*동의대학교 응용소프트웨어공학과, **과학기술연합대학원대학교, ***한국전자통신연구원
e-mail : *aoikazto@naver.com, jubin@etri.re.kr, yongin.kwon@etri.re.kr*

Profiling-based ArmCL Optimal Schedule Search for Single-ISA Heterogeneous Multi-Core Architectures

*Joo Hyoung Cha, **Jubin Lee, ***Yongin Kwon
*Dong-Eui University
**University of Science and Technology
***Electronics and Telecommunications Research Institute

Abstract

ARM Cortex-A 아키텍처에서 머신 러닝 연산을 가속화하기 위해 ArmCL[1]을 제공하고 있다. ArmCL은 이기종 코어 구조에서 Big 코어의 수만큼 쓰레드를 생성하여 병렬 연산하고 있어 모든 CPU 자원을 활용하지 못한다. 본 논문에서는 Big 코어만 활용하는 기존 ArmCL 스케줄러를 대체하여 모든 CPU 자원을 활용하는 스케줄러 구현과 최적화 방법을 소개한다. 실험 결과 기존 ArmCL과 비교하였을 때 연산 성능이 최대 39% 향상이 되는 것을 확인했다.

I. 서론

최근 머신러닝의 보편화로 인해 학습과 추론을 가속하기 위해서 GPU, NPU 등 여러종류의 하드웨어가 사용되고 있다. 하지만 모바일 디바이스나 임베디드 시스템에서는 전력 및 비용 등의 문제로 CPU가 널리 활용되고 있고, CPU에서 효율적으로 머신 러닝 연산을 수행하기 위한 연산 알고리즘 및 소프트웨어 연구도 활발하게 진행 중이다.

Arm은 고성능 Cortex-A CPU의 낮은 전력효율 문제를 해결하기 위해 단일 ISA를 가지는 두 종 이상의 이기종 코어를 함께 집적하는 big.LITTLE 솔루션과 이를 제어하는 big.LITTLE MP를 개발하였다. big.LITTLE MP는 리눅스 커널 단에서 구현된 GTS(Global Task Schedule)를 활용하여 측정된 코어의 부하를 기반으로 특정 쓰레드를 다른 코어로 마이그레이션한다[4]. 대표적인 big.LITTLE 구조의 SoC는 Hi-Silicon Kirin, Samsung Exynos, Qualcomm Snapdragon이 있고, 이를 효율적으로 활용하기 위한 연구와 논의가 진행되고 있다[5].

ArmCL(Arm Compute Library)[1]은 Arm에서 머신 러닝 연산을 가속하기 위하여 개발된 오픈소스 프로젝트로, CPU에서의 머신 러닝 연산 시 ARM NEON기술을 활용도록 구현되어 있다. NEON은 CPU에서 멀티미디어 및 신호 처리 응용 프로그램의 성능을 향상시키기 위해 개발된 SIMD(Single Instruction Multiple Data) 아키텍처로, ARMv7 이후 도입되어 머신 러닝 가속에도 효율적이다. big.LITTLE 구조에서의 ArmCL은 Big 코어의 수만큼의 쓰레드를 생성한 뒤 런타임에 GTS 통하여 Big 코어로 마이그레이션한다. 머신 러닝 연산에 따라 실행에 적합한 쓰레드 수나 코어 종류를 선택하지

못하기 때문에, 성능 최적화에 한계가 있다.

본 논문에서는 프로파일 기반으로 최적의 스케줄을 탐색하여 최적의 쓰레드 수 및 쓰레드 마다 동작할 코어의 종류를 미리 결정할 수 있도록 한다. 더 나아가 이기종 코어에 적합한 쓰레드별 작업 사이즈를 분배하기 위해 새로운 스케줄러 구현을 추가하여 최적화하는 방법을 제안한다. 해당 구현의 효과를 검증하기 위해 이기종 코어 구조를 가진 RK3399Pro에서 동작하는 AlexNet[2]과 VGG16[3] 모델의 최적화 전/후 성능을 비교한다.

II. ArmCL 구조

ArmCL은 컴파일 타임과 런타임에 결정되는 요소가 있다. 컴파일 타임에는 사용할 멀티 쓰레딩 라이브러리, CPU 아키텍처 종류 및 빌드 옵션 지정이 가능하다. 런타임에는 ArmCL 초기화와 그래프 검증, 연산 과정과 같이 세 가지 과정이 존재한다.

초기화 과정에는 CPU의 코어 수, 모델명, 캐시 구조 등의 정보를 수집하여 스케줄러에서 활용하는 쓰레드를 생성한다. 생성되는 쓰레드의 수는 최소 빈도 코어의 수만큼 쓰레드를 생성하게 된다. 만약 Big 코어가 4개, Little 코어가 2개가 존재한다면, 쓰레드 수는 4개가 아닌 2개가 된다.

그래프 검증 과정에는 가중치 데이터를 메모리 적재하고 인공지능 모델을 머신 러닝 연산 요소인 Kernel로 변환한다. 만약 컨볼루션 레이어라면, 표. 1과 같이 3종의 알고리즘 중 하나를 선택하게 되는데, 해당 컨볼루션 연산의 입출력 레이아웃과 가중치의 가로, 세로 사이즈 등이 고려되는 요소이다[6]. Kernel의 GEMM 연산은 세부적으로 1개 이상의 구현체가 존재하는데, 각 구현체는 행렬연산 기본단위 크기에 따라 NEON을 활용하는 Assembly 언어로 구현되어 있다. 구현체를 선택하기 위해 Kernel은 수집된 CPU 모델의 종류와 실행 시간을 예측하여 결정한다.

표 1. 컨볼루션 연산 최적화 옵션

컨볼루션 알고리즘	GEMM Kernel의 구현체
Winograd	Sgemm 8*6
	Sgemm 8*12
Gemm General	mla 8*4
	mla 4*24
Gemm Direct	mla 6*16

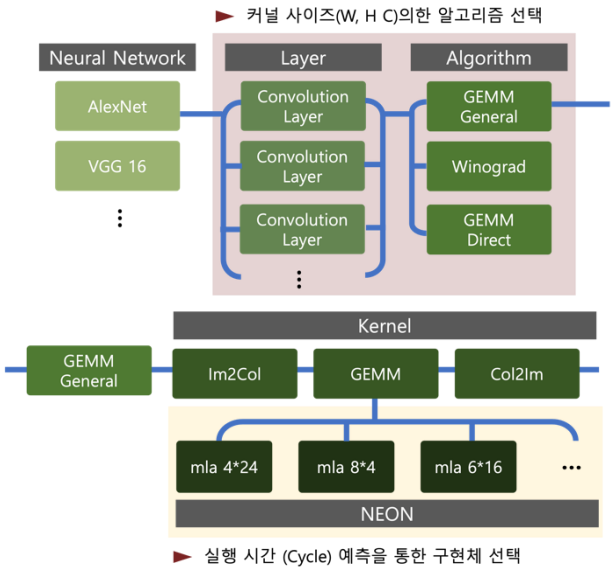

그림 1. ArmCL 내부 구조도

연산 과정에는 미리 적재된 가중치, 입력 텐서, Kernel을 스케줄러에 전달하여 병렬 연산을 수행한다[7]. 또한 스케줄러는 Kernel의 구현체에서 파티셔닝 전략과 작업 사이즈 정보를 수집하여 각 쓰레드에 균등하게 작업량을 분배한다.

III. 프로파일 기반 최적 스케줄 탐색

프로파일 기반 탐색 기법은 실행 가능한 모든 파티션 조건을 미리 생성하여 최적 해를 탐색한다. 모든 경우를 탐색하여 최적 해를 찾는 과정은 표. 2 과 같이 경우의 수가 대량으로 발생하여 연산 당 수십 시간이 소요된다. 따라서 본 논문에서는 표. 3 의 Q 및 R 값의 변화를 통해 탐색 빈도를 조절하고 코어 별 파티셔닝과 여러 컨볼루션 연산 옵션에 대해서만 프로파일링을 진행하는 컨볼루션 최적화를 통해 주어진 시간 내에 최적의 결과를 찾을 수 있도록 한다.

표 2. 전체 프로파일의 경우의 수

$$core = [count(little) \quad count(big) \quad count(big.Little)]$$

$$w = work\_size\_of\_kernel$$

$$partition_{totalcase} = \sum_{k=1}^{len(kernel_{list})} \sum_{n=1}^{len(core)} core_n H_w$$

표 3. 최적 스케줄 탐색 의사코드

```
1: conv := [‘winograd’, ‘gemm_asm’, ‘gemm’]
2: matrix := [‘sg8x4’, ‘sg8x12’,
3:            ‘mla8x4’, ‘mla4x24’, ‘mla6x16’]
4: core := [‘big’, ‘little’, ‘big.LITTLE’]
5: ops_list := get_all_kernel_api()
6: Q := # 작업 사이즈의 구간과 간격을 나누는 상수
7: R := # 최대, 최소 작업 사이즈 계산을 위한 상수
8: skip_convolution:= On/Off # 시간을 위한 최적화 생략 변수
9: function find_best_partition(work_size, c)
10:   step := work_size / count(c) / Q
11:   min, max := step / R, step * R
12:   w := (max - min) / step
13:   best := partition_info() # 파티셔닝 결과를 저장
14:   best_time := +inf
15:   for partition ∈ count(c)H_w # H는 중복조합
16:     remove_cache()
17:     start_time := measure_time()
18:     run_ops(kernel_ops, partition * step, c)
19:     if best_time > measure_time() - start_time
20:       best = partition
21:   return best
22:
23: function tuner_entry_point()
24:   for kernel_ops ∈ [ops_list], c ∈ [core]
25:     input := get_tensor()
26:     work_size := compute_work_size(input, kernel_ops)
27:     if is_convolution(kernel_ops) and !skip_convolution
28:       for mode ∈ [conv], m ∈ [matrix]
29:         best = min(find_best_partition(work_size, c), best)
30:     else
31:         best = min(find_best_partition(work_size, c), best)
```

코어 별 파티셔닝은 CPU 코어 마다 할당되는 작업 사이즈를 제어하여 성능을 최적화한다. 표. 3의 find_best_partition 함수는 작업 사이즈를 달리하여 생성된 코드를 프로파일하여 최적의 파티션을 찾도록 해준다.

표. 3의 27~31라인에서 실행되는 컨볼루션 최적화는 표 1의 컨볼루션 알고리즘과 GEMM Kernel 구현체의 조합 중 최적을 찾는 것으로 모든 조합에 대해 프로파일하여 최적의 조합을 도출해낸다.

표 4. 스케줄 최적화 알고리즘 옵션

	코어 별 파티셔닝	컨볼루션 최적화
기본 ArmCL	X	X
스케줄링 최적화	O	X
추가 컨볼루션 스케줄링 최적화	O	O

Ⅳ. 실험 환경

실험은 Asus Tinker Edge R 보드에서 진행하였으며 보드에 탑재된 SoC는 RK3399Pro로, big.LITTLE 구조의 Arm Cortex-A CPU 아키텍처를 포함하고 있다. RK3399Pro는 A72(Big) 코어 2개와 A53(Little) 코어 4개가 있는 구조이다[8]. A72와 A53은 각각 1.8Ghz, 1.4Ghz로 고정하였으며, 코어에 1개씩 쓰레드를 생성하여 할당한다.

머신러닝 모델로는 ImageNet 데이터셋에 대한 AlexNet[2]과 VGG16[3]으로 실험을 진행하였으며 ArmCL의 버전은 22.05를 사용하였다. 성능에 영향을 주는 DVFS(Dynamic Voltage Frequency Scaling)를 제어하기 위해 CPU 주파수를 고정하였으며, 쓰레드가 특정 코어에서 연산하도록 하였다. 또한 캐시 메모리로 인한 성능 오차를 최소하기 위해 매 측정 마다 캐시 메모리를 삭제한다.

성능 측정 방식은 8회 반복하여 각 시행 마다 시간을 측정하였으며, 성능을 평가하기 위해 최대, 최소를 제외한 절사 평균 값을 활용하였다. 또한 Kernel 마다 최적화 시간을 1시간 기준으로 하기 위해 표 1의 변수 Q는 10, R은 2로 진행하였다.

Ⅴ. 결과

본 문은 VGG16 과 AlexNet 의 각 커널에 대해서 적합한 스케줄링, 결과를 정리한 내용이다. 표. 4 의 3 가지 방식의 스케줄 결과를 통해 소요된 시간, 성능을 비교 한다.

먼저 스케줄 과정에 발생한 탐색 시간을 정리한다. 기본 ArmCL 은 최적화를 하지 않으므로 0 이며, 스케줄링 최적화는 AlexNet 과, VGG16 에서 평균적으로 약 4-5 시간이 소요되었다. 추가 컨볼루션 스케줄링은 모든 컨볼루션을 탐색해야하므로 약 100~135 시간이 소요되었다.

표 5. AlexNet과 VGG16 최적화 전/후 결과

AlexNet 추론/탐색 시간		
	추론 시간 (단위 : ms)	탐색 시간 (단위 : 초)
기본 ArmCL	115.521	0
스케줄링 최적화	84.853	16,170
추가 컨볼루션 스케줄링 최적화	75.794	378,666
VGG16 추론/탐색 시간		
	추론 시간 (단위 : ms)	탐색 시간 (단위 : 초)
기본 ArmCL	750.080	0
스케줄링 최적화	484.924	12,483
추가 컨볼루션 스케줄링 최적화	460.326	486,039

표. 5 는 기본 ArmCL 방법과 스케줄링 최적화, 추가 컨볼루션 스케줄링 최적화 방법을 사용하여 AlexNet 과 VGG16 에 모델 추론 시간을 정리한 것이다. AlexNet 과 VGG16 모두 기본 ArmCL 의

추론 시간이 가장 많이 소요되었다. 스케줄링 최적화만을 진행하였을 , 추론 시간이 AlexNet 에서 84.85ms 로 약 30% 개선 되었고, VGG16 에서 484.92ms 로 약 35% 개선이 되었다. 추가 컨볼루션 스케줄링 최적화 결과 추가적 때인 개선이 되었지만 스케줄링 최적화 방법과 비교하였을 때는 미미하였으며, 최적화에 약 30 배 정도의 탐색 시간이 더 걸렸다.

표 6. AlexNet의 레이어 별 성능 변화 (단위 : ms)

레이어		Case 1	Case 2	Case 3
01	Convolution	21.252	13.550	11.116
02	Convolution	32.924	19.317	13.989
03	Convolution	8.306	6.002	6.002
04	Convolution	8.348	6.074	5.362
05	Convolution	7.167	5.137	4.148
06	FullyConnected	37.449	34.699	34.699
07	SoftMax	0.026	0.026	0.026
Total		115.521	84.853	75.794

표 7. VGG16 의 레이어 별 성능 변화 (단위 : ms)

레이어		Case 1	Case 2	Case 3
01	Convolution	19.671	14.805	8.132
02	Convolution	130.160	100.816	97.216
03	Convolution	53.261	33.163	32.688
04	Convolution	84.458	49.151	49.151
05	Convolution	35.753	19.925	19.925
06	Convolution	59.937	32.874	32.874
07	Convolution	61.286	33.748	33.748
08	Convolution	37.561	18.982	17.137
09	Convolution	66.332	37.353	31.502
10	Convolution	66.150	37.890	31.737
11	Convolution	19.445	11.424	11.424
12	Convolution	19.373	11.438	11.438
13	Convolution	19.320	11.586	11.586
14	FullyConnected	77.340	71.736	71.736
15	SoftMax	0.026	0.026	0.026
Total		750.080	484.924	460.326

표. 6, 7 은 두 모델의 각 최적화 알고리즘과 레이어에 따른 성능 변화를 나타내었다. Case 1, 2, 3 은 표 4 의 스케줄 최적화 알고리즘 옵션이며 각각 기본 ArmCL, 스케줄링 최적화, 추가 컨볼루션 스케줄링 최적화를 의미한다.

각 최적화 알고리즘은 두 모델의 레이어에 따라 순차적으로 적용된다. 특정 레이어에서 스케줄링 최적화와 추가 컨볼루션 스케줄링 최적화의 결과가 동일한 경우가 존재한다. 이는 컨볼루션 최적화를 통한 최적의 조합의 결과와 기본 ArmCL 의 알고리즘과 구현체가 동일하다는 의미이다.

실험 결과 이기종 코어를 위한 스케줄러와 컨볼루션 알고리즘, 구현체를 탐색한 결과 AlexNet 과 VGG16 에서 최적화 후 성능이 최대 39% 개선되었다.

VI. 결론

본 논문에서 기존 ArmCL 의 스케줄러를 수정하여 이기종 코어 구조에 적합한 파티셔닝 할 수 있는 스케줄러를 구현하였다. 스케줄링 결과, 시간 대비 효율은 스케줄링 최적화가 효율적인 것을 확인하였다. 추가 컨볼루션 스케줄링 최적화는 향후 하이퍼 파라미터 튜닝과정을 추가한다면 최적화 시간을 줄일 수 있을 것이라고 생각한다.

또한 스케줄 결과에서 일부 Kernel 은 이기종 코어 구조가 아닌 단일 코어 구조가 성능이 더 좋은 결과가 나타났다. 그래서 다양한 시나리오를 작성하여 모델과 장비를 활용하여 실험이 필요하다.

추가적으로 DVFS 와 쓰레드를 특정 코어에 할당하는 기법은 관리자 권한이 필요하므로 모바일 스마트폰 기기에 적용 하는 것은 어려움이 있다. 그래서 추가적인 연구를 통해 복잡한 이기종 코어 구조와 일반 사용자 권한에서 동작하는 스케줄러 구현이 필요하다.

사사

이 논문은 2022 년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임 (No.2018-0-00769,인공지능 시스템을 위한 뉴로모픽 컴퓨팅 SW 플랫폼 기술 개발)

참고문헌

[1] ArmCL, https://github.com/ARM-software/ComputeLibrary

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." NIPS 2012

[3] Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556. 2014

Sep 4.
[4] 윤석훈, 문영일, 구본규, 유희재 저, 코드로 알아보는 ARM 리눅스 커널, 제이펍, 2018.
[5] W. Seo, D. Im, J. Choi and J. Huh, "Big or Little: A Study of Mobile Interactive Applications on an Asymmetric Multi-core Platform," 2015 IEEE International Symposium on Workload Characterization, 2015, pp. 1-11,
[6] Maji, Partha, et al. "Efficient winograd or cook-toom convolution kernel implementation on widely used mobile cpus." 2019 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2). IEEE, 2019.
[7] de Prado, Miguel et al. "Automated Design Space Exploration for Optimized Deployment of DNN on Arm Cortex-A CPUs." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 40 (2021): 2293-2305.
[8] RK3399Pro DataScheet, https://rockchip.fr/RK3399Pro%20datasheet%20V1.1.pdf