Journal of The Institute of Electronics and Information Engineers Vol.60, NO.7, July 2023 https://doi.org/10.5573/ieie.2023.60.7.40
ISSN 2287-5026(Print) / ISSN 2288-159X(Online)

논문 2023-60-7-5

이기종 멀티코어 CPU에서 프로파일 기반 딥 러닝 연산 최적화 기법

(Profile-based Optimization for Deep Learning on Heterogeneous Multi-core CPUs)

차 주 형*, 권 용 인**, 이 제 민**

(Joo Hyoung Cha, Yongin Kwon©, and Jemin Lee)

요 약

최근 임베디드 환경에서 딥 러닝을 적용하고자 하는 요구가 증가하고 있다. 임베디드와 같은 제한적인 환경에서 딥 러닝 연산을 효율적으로 수행하기 위해서 Arm의 big.LITTLE과 같은 이기종 멀티코어 CPU 아키텍처가 널리 활용되고 있다. Arm은 딥 러닝 연산을 최적으로 수행하기 위해 Arm Compute Library(ACL)를 제공하고 있지만, big.LITTLE 구조를 가진 하드웨어의 잠재력을 충분히 활용하지는 못하고 있다. 본 논문은 각 하드웨어에 최적인 실행 커널과 스케줄을 자동으로 결정하기 위한 프로파일 기반 탐색 방법을 제안한다. 실험은 Tinker Edge R, Odroid N+, Snapdragon 865 HDK 보드에서 AlexNet, VGG16, MobileNetV2, GoogleNet 모델을 대상으로 진행하였으며, 모든 경우에서 제안된 방법이 기존의 방법보다 최대 266% 성능 향상을 보임을 확인하였다. 본 연구의 결과를 통해 임베디드 기기에서 저비용, 저전력, 고성능의 딥 러닝 수행이 가능할 것으로 기대한다.

Abstract

Recently, there has been a growing demand to apply deep learning in embedded environments. In constrained embedded environments, heterogeneous multicore CPU architectures like Arm's big.LITTLE are widely utilized to efficiently carry out deep learning computations. Although Arm provides Arm Compute Library (ACL) for optimal deep learning operations, it does not fully leverage the potential of hardwares with the big.LITTLE structure. This paper proposes a profile-based search method for automatically determining the optimal execution kernel and schedule for each hardware. Experiments were conducted on Tinker Edge R, Odroid N+, and Snapdragon 865 HDK boards using AlexNet, VGG16, MobileNetV2, and GoogleNet models. In all cases, the proposed method improved performance up to 266% compared to existing methods. Through the results of this research, we expect to enable cost-effective, low-power, and high-performance execution of deep learning in embedded devices.

Keywords: Deep learning, Heterogeneous multi-core, Optimization, big.LITTLE, Embedded

Ⅰ. 서 론

이미지 분류, 객체 인식, 자연어 처리 등에서 딥 러닝을 활용한 추론의 정확도가 점점 높아짐에 따라, 자율주행 자동차, 인공지능 CCTV 등 다양한 분야에 적용하려는 움직임이 활발하다. 이러한 시장의 요구로 인해 GPU 기술 또한 꾸준히 발전하여 서버에서는 딥 러닝 연산을 빠르고 효율적으로 처리하여 학습과 추론이 가

*비회원, 동의대학교 응용소프트웨어공학과(Department of Applied Software Engineering, Dong-Eui University)
**비회원, 한국전자통신연구원(Electronics and Telecommunications Research Institute)
© Corresponding Author(E-mail : yongin.kwon@etri.re.kr)
※ 이 논문은 2023 년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임 (No. 2022-0-00454, 스마트 엣지 디바이스 SW 개발 플랫폼 개발).
Received ; April 14, 2023 Revised ; April 24, 2023 Accepted ; May 16, 2023

능하다. 하지만 실시간성이 중요한 임베디드 환경에서는 제한된 자원으로 인해 딥 러닝 추론조차 처리하기 쉽지 않다.

그럼에도, 서버에서의 딥 러닝 연산 처리 시 발생하는 보안, 네트워크, 지연시간 등의 문제 해결을 위해, 임베디드 환경에서의 딥 러닝 연산이 필요한 상황이 있다. 또한, 임베디드 환경에서는 전력과 비용 문제로 인해 GPU나 딥 러닝 전용 가속 하드웨어 없이 CPU만으로 연산을 처리해야 할 수도 있다.

임베디드 시스템은 제한된 전력 환경에서 높은 효율 요구를 가지므로 단일 ISA 이기종 멀티코어 CPU를 활용하여 문제를 해결하고자 하는 노력이 있다. 그중에서도 Arm의 big.LITTLE은 저전력/고효율 코어와 고전력/고성능 코어 2종류를 사용하여 한정된 컴퓨팅 자원과 최소한의 전력으로 높은 성능을 발휘한다[1]. 특히 Arm의 Cortex-A CPU는 SIMD[2]기술인 NEON을 통하여 여러 데이터 요소에 동시에 작업을 수행하는 병렬 처리 연산이 가능하여 딥 러닝 연산에 유리하다.

Arm은 자사의 CPU와 GPU에서 딥 러닝 연산 처리를 가속하기 위한 연산 라이브러리로, ACL(Arm Compute Library)[3]을 제공한다. Arm의 ISA(Instruction Set Architecture)를 공유하는 여러 종의 Cortex-A CPU 모델들은 ACL을 통하여 딥 러닝 연산을 처리할 수 있지만, 규칙 기반으로 실행 코드와 스케줄을 선택하도록 구현된 ACL은 종종 최적이 아닌 실행을 선택한다. CPU 모델과 구현에 따른 특성 차이, 연산의 종류와 형태 등에 따라 최적의 코드와 실행 방법이 달라지지만, 규칙 기반 실행은 CPU 코어를 충분히 활용하지 못하여 성능을 저하한다.

본 논문은 big.LITTLE 구조의 CPU에서 ACL이 각 하드웨어와 딥 러닝 연산에 대해 최적의 실행 코드와 스케줄을 찾아낼 수 있도록 프로파일 기반 탐색을 통한 최적화 방법을 제안한다. 또한, 효율적 탐색을 위한 알고리즘으로는 그리드 탐색[4], 랜덤 탐색[4], 베이지안 최적화[5]를 적용하고 그 결과를 비교한다.

본 논문의 구성은 다음과 같다. Ⅱ장에서 Arm big.LITTLE 및 ACL의 구조에 대해 설명한다. Ⅲ장에서 제안하는 최적화 기법에 대해 설명하고, 이를 검증하기 위한 실험 결과를 Ⅳ장에서 분석한다. 마지막으로 Ⅴ장에서 결론을 맺는다.

Ⅱ. 배 경

1. Arm big.LITTLE

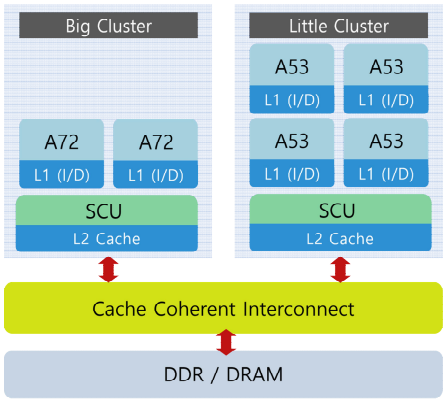

그림 1. RK3399Pro[6]의 big.LITTLE 아키텍처 구조
Fig. 1. big.LITTLE Architecture of RK3399Pro.

전력 소모량을 최적화하기 위해 전압과 주파수를 동적으로 조절하는 DVFS(Dynamic Voltage Frequency Scaling)는 프로세서의 성능이 저하될 수 있고 스케일링 오버헤드가 발생한다. 이러한 문제점을 해결하기 위해 Arm의 big.LITTLE 기술은 저전력/고효율 코어(LITTLE)와 고전력/고성능 코어(big)를 함께 사용하고, 작업의 부하에 따라 적절한 코어를 선택하여 처리함으로써 전력 소모를 줄이면서 성능 저하를 최소화한다. 그림 1과 같이 동종의 코어를 두 개 이상 사용하여 클러스터로 구성할 수 있으며, 고성능 스마트폰에서 사용되는 AP(Application Processor)의 경우 세 개의 클러스터를 구성하여 운용하기도 한다.

ARM의 big.LITTLE 구조에서는 코어 간의 스케줄을 효율적으로 관리하기 위해 HMP(Heterogeneous Multi-Processing)를 사용하며, 다음과 같은 방식으로 성능과 전력 소모량을 최적화한다.

■ 작업 스케줄링: 운영체제의 스케줄러는 쓰레드를 적절한 클러스터로 할당하기 위해 작업의 우선순위, 코어 사용률, 전력 관리 정책 등을 고려한다. 이를 위해 프로세서에는 성능 모니터링 레지스터(PMRs)가 있어, 각 코어의 성능과 부하 상태를 실시간으로 파악할 수 있다. 쓰레드가 특정 클러스터에 할당되어 실행되고 있더라도 지속적인 성능 모니터링을 통해 다른 코어로 마이그레이션될 수 있다.

■ 동적 작업 마이그레이션: HMP는 다양한 스케줄링 알고리즘(GTS[7], EAS[8])에 의해 각각의 목적을 달성하기 위한 마이그레이션을 수행한다. 마이그레이션은

동일 클러스터 내에서 이루어지는 경우와 다른 클러스터 사이에서 이루어지는 경우로 나눌 수 있다. 동일한 클러스터 내에서의 마이그레이션은 그림 1의 L2 캐시를 코어 사이에서 공유하기 때문에 캐시 지역성을 유지할 수 있다. 따라서 마이그레이션 비용이 적으며 주로 클러스터 내에서 부하의 균형을 유지하는 데 사용된다.

작업의 연산 부하가 현재 코어에서 처리하기에 적합하지 않거나 더 효율적으로 처리할 수 있는 코어가 다른 클러스터에 있을 때, 클러스터 사이에서의 마이그레이션이 이루어진다. 이 경우, Cache Coherent Intercon- nect(CCI)[9]를 통해서 현재 작업 정보를 전달해야 하므로 동일한 클러스터에서 이전하는 것보다 비용이 크다. 그럼에도 불구하고 성능 향상과 에너지 효율이 향상하기 때문에 현재 부하의 특성, 코어의 성능, 시스템의 상태 등 다양한 요소를 고려하여 스케줄링 알고리즘이 마이그레이션을 판단하게 된다.

■ 병렬 처리: HMP는 시스템 작업의 연산 부하에 따라서 big 코어와 LITTLE 코어를 동시에 사용할 수 있다. 따라서, 충분한 수의 쓰레드를 생성하면 여러 클러스터를 동시에 사용하는 등의 최적화가 가능하다. 다만, 공유하고 있는 캐시의 특성을 잘 반영하여 같은 클러스터 상에서 공유하는 L2 캐시의 히트율을 극대화해야 하며, 메모리 대역폭의 제약사항도 고려할 필요가 있다.

2. Arm Compute Library (ACL)

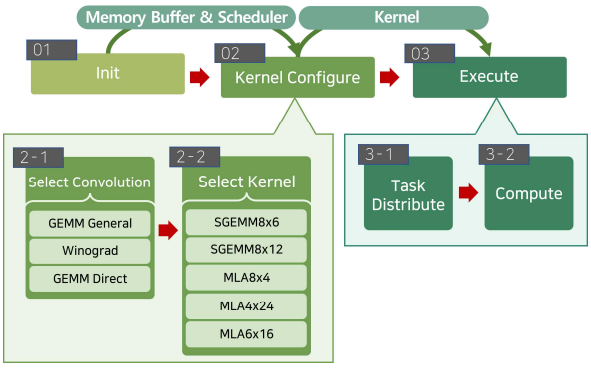

그림 2. 딥 러닝 연산을 위한 ACL 실행 과정
Fig. 2. Procedure of ACL for deep learning.

가. ACL 구조

ACL[3]은 Arm 프로세서를 위한 오픈소스 소프트웨어 라이브러리로, 컴퓨터 비전 및 기계 학습을 위한 기능을 제공하며, Arm Cortex-A와 Arm Mali GPU와 같은 Arm 아키텍처 기반의 시스템에서 최적화된 성능을 달성하기 위해 구현되어 있다. 특히, Arm CPU의 효율적 활용을 위해 NEON을 지원하고 있으며, 이를 활용하여 딥 러닝의 추론 연산을 빠르고 효율적으로 처리할 수 있다.

ACL은 C++ 언어로 작성되어 있으며, 딥 러닝 모델의 가속을 위해 필요한 다양한 함수와 API를 제공한다. 그중에는 데이터 전처리 및 후처리를 위한 함수가 포함되며, TensorFlow Lite[10]와 ONNX 런타임[11]과 같은 딥 러닝 프레임워크와 통합되어 Arm 기반 하드웨어에서 딥 러닝 모델을 쉽게 실행할 수 있다.

ACL의 컴파일 과정에서는 사용할 멀티스레딩 라이브러리(pThread, OpenMP)[12], 운영체제의 종류(Andr- oid, Linux), CPU 아키텍처(armv8.2-a-sve, armv8a, 등) 종류, SIMD 지원 등의 정보가 수집 및 선택되어 고정된다.

ACL을 통하여 딥 러닝 연산을 실행하는 과정은 그림 2와 같이 초기화, 커널 구성, 실행의 세 단계로 구성된다. 먼저, 초기화 단계에서는 ACL 함수와 객체를 초기화하고, 메모리 버퍼, 텐서, 커널 및 관련 자원을 설정한다. 커널 구성 단계에서는 하드웨어 및 아키텍처 정보를 고려하여 연산에 필요한 커널을 결정하고 구성하며, 쓰레드를 생성한다. 또한, 가중치 데이터를 메모리에 적재하며, 생성된 쓰레드는 이후의 실행에서 재사용된다. 마지막으로, 실행 단계에서는 메모리에 적재된 가중치와 구성된 커널 쓰레드를 사용하여 딥 러닝 연산을 수행한다.

나. 커널 구성

ACL은 딥 러닝 모델의 다양한 레이어와 함수들을 효율적으로 실행할 수 있도록 지원하기 위해 모델의 각 레이어를 개별적으로 처리하며, 레이어 간의 데이터 전달을 관리한다.

CPU에서 딥 러닝 연산을 수행하기 위한 최소 연산 과정은 다음과 같다. 먼저, 입력 데이터를 메모리로부터 특정 레지스터에 읽어 들인다. 그 다음, 딥 러닝 연산을 수행하여 필요에 따라 결과를 메모리에 다시 저장한다. 대부분의 딥 러닝 연산의 크기는 CPU 레지스터의 크기보다 훨씬 크기 때문에 이 과정을 여러 번 반복 수행한다. 이러한 최소단위의 연산 과정을 커널이라 부른다.

ACL에는 동일한 연산을 수행하는 커널이 다양하게 구현되어 있다. 해당하는 ISA를 지원하는 모든 CPU 모델들은 어떤 커널을 실행하더라도 연산 결과는 항상 동

일하다. 하지만 연산 형태(연산 종류, 입/출력 및 가중치의 사이즈 등)에 따라 성능이 달라질 수 있으며, ACL에서는 규칙 기반으로 적절한 커널을 선택하도록 구현되어 있다. 특히, 딥 러닝의 대표적인 연산 중 하나인 컨볼루션(Convolution)의 최적 연산을 위해 표 1과 같이, ACL은 GEMM General[13], Winograd[14], GEMM Direct[15] 등 다양한 컨볼루션 방법을 제공하고 있다. 선택된 컨볼루션 방법에 따라 연산 종류 및 순서가 결정되고, 각 연산은 다시 여러 커널 중 하나를 조합하여 수행되게 된다.

표 1. 컨볼루션 레이어의 방법 종류
Table 1. type of algorithms for convolution layer.

	Winograd[14]	General[13]	Direct[15]
조건	Kernel Size: 3x3, 1x3, 3x1, 5x1, 1x5, 7x1, 1x7	없음	DataLayout: NHWC
Pre	Transpose	Im2Col	skip
Ops	GEMM	GEMM	GEMM
Post	Transpose	if NHWC then skip	

선택된 컨볼루션 방법과 관계없이, 컨볼루션 연산은 GEMM(GEneral Matrix to Matrix multiplication) 커널을 항상 수행하게 되어있다. ACL은 C++로 구현한 Naive 커널부터, 각 코어의 아키텍처와 캐시 크기까지 고려해 최적화된 어셈블리어 커널까지 다양하게 제공한다. 커널 구성 단계에서는 사용 가능한 모든 커널을 탐색한 뒤, 각각의 경우에 대한 실행 사이클을 예측하고 최선의 커널을 선택하는 과정을 거친다.

표 2. GEMM 커널의 종류
Table 2. type of GEMM Kernel.

Name	Kernel Size (H, W)	Memory Arrange
SGEMM	8 x 6	Interleaved[16]
	8 x 12	Interleaved
MLA	8 x 4	Hybrid Indirect[17]
	4 x 24	Hybrid Indirect
	6 x 16	Hybrid Indirect

표 2는 본 논문의 실험환경에서 주로 사용된 부동소수점 GEMM 연산 커널의 목록을 나타낸다. GEMM 커널은 메모리 최적화 기법에 따라 크게 Interleaved[16]와 Hybrid Indirect[17] 기법으로 나뉜다. Interleaved는 연산 중, 캐시 구조에 적합하도록 메모리 구조를 변경하는 최적화 기법을 사용한다. 반면에, Hybrid Indirect는 메모리 구조를 변경하지 않지만, 간접 버퍼를 도입하여 메모리 오버헤드를 감소하는 기법을 사용하여 최적화한다. 각 기법의 커널은 세부적으로 최소단위 연산 크기에 따라 8x6, 6x16 등으로 구분되어 있다.

다. 쓰레드 생성

ACL은 커널 구성 단계에 다중 쓰레드를 생성하여 여러 코어에서 커널이 병렬 처리되도록 한다. ACL은 생성되는 쓰레드의 수를 정하기 위해 Alg (1)을 사용하여 동일 CPU 모델별 코어의 개수 중 가장 작은 수 만큼의 쓰레드를 생성한다.

$$\min(\mathrm{count}(\mathrm{CPU}_{\mathrm{Model1}}), \mathrm{count}(\mathrm{CPU}_{\mathrm{Model2}}), \ldots) \qquad \text{Alg (1)}$$

예를 들어, 그림 1의 경우 Big 클러스터에 코어가 2개, Little 클러스터에 코어가 4개 존재하므로, 쓰레드 수는 $\min(2,4)=2$ 가 된다.

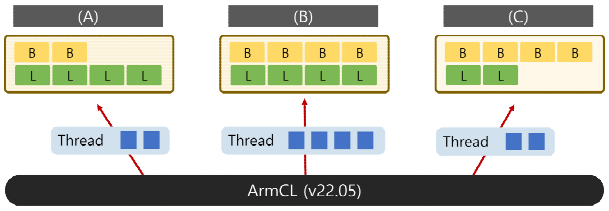

그림 3. ACL 스케줄러의 쓰레드 생성 예
Fig. 3. Example of thread generation in ACL Scheduler.

그림 3은 big.LITTLE 구조에서 가질 수 있는 여러 경우에 따른 생성되는 쓰레드 개수의 예를 나타낸다. 그림 3(A)의 경우, big 코어의 수 만큼 쓰레드가 생성되므로 big 클러스터를 최대한 활용하여 딥 러닝 연산을 수행할 수 있으나 LITTLE 클러스터에서는 모든 코어를 충분히 활용하지 못한다. (B)의 경우 각 클러스터를 단독으로 활용 시 최고 성능을 발휘할 수 있으며, (C)의 경우 2개의 쓰레드로는 모든 big 코어를 활용하지 못한다. 결과적으로 ACL이 제공하고 있는 스케줄러는 어떠한 경우에도 모든 클러스터의 모든 코어를 동시에 활용하여 딥 러닝 연산을 수행할 수 없다.

라. 실행

커널 구성 단계에서 최소단위 연산을 위한 커널이 선택되고 특정 개수의 쓰레드가 생성이 되면, 실행 단계에서는 그림 4와 같이 스케줄러에 의해 각 레이어 입/

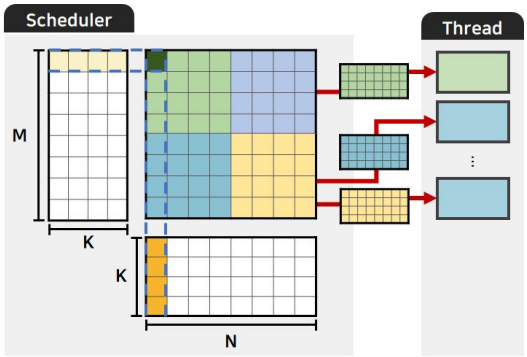

그림 4. GEMM 연산 분할 및 스케줄링 예제
Fig. 4. Partitioning and scheduling of GEMM.

출력을 커널 크기로 분할하여 작업을 생성한다. ACL은 생성된 작업들을 쓰레드에 균등하게 분배하여 병렬 연산을 수행하도록 구현되어 있다.

각 쓰레드가 어떠한 클러스터에서 실행될지는 지정되어 있지 않으며, HMP에 의해 동적으로 제어된다. 커널 구성 단계에서 정해진 쓰레드의 수는 변경되지 않기 때문에 유휴 코어가 있다고 할지라도 활용할 수 없다. 만약 전체 클러스터의 코어 수만큼 쓰레드를 생성한다고 하더라도, 작업을 균등하게 배분하기 때문에 고성능 코어에서의 연산이 더 빨리 끝나 여전히 유휴 연산 자원이 생기는 문제점이 있다.

Ⅲ. 제안 방법

1. 사용자 정의 커널 구성 및 스케줄링

ACL은 커널 구성 단계에서 규칙 기반으로 연산 방법(컨볼루션 방법 등)을 결정한다. 이에 따라 커널을 선택하여 구성하며, 생성되는 쓰레드의 수는 Alg (1)에 의해서 결정된다. 이 경우, 최적화되지 않은 커널이 구성되거나 코어 자원을 충분히 활용하지 못할 수 있다.

본 연구에서는 연산 방법 및 커널 구성을 사용자가 직접 정의할 수 있는 방법을 제안한다. 또한, 생성된 쓰레드에 작업을 임의로 분배하거나 사용자 작성 알고리즘에 의해 분배할 수 있도록 구현하고, 각 쓰레드가 HMP에 의해 마이그레이션 되지 않고 특정 클러스터에서만 동작하도록 사용자가 지정할 수 있도록 하였다.

하지만 컨볼루션 연산마다 대략 2×10^{12} 개가 넘는 경우의 수가 존재하기 때문에, 사용자가 그중에서 최적을 찾기는 불가능하다. 따라서 본 연구에서는 그림 5의 최적 커널 구성 및 스케줄 탐색을 위한 시스템을 제안하여, 다양한 탐색 알고리즘에 의해 자동으로 최적에 가까운 커널 구성과 스케줄을 찾는다.

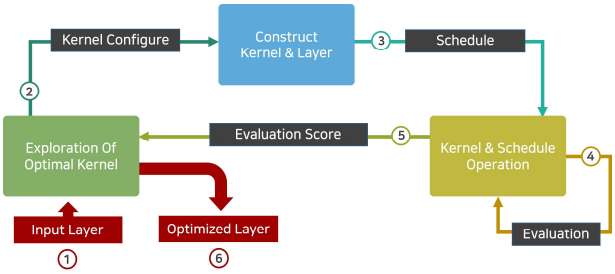

그림 5. 최적 커널 및 스케줄을 탐색
Fig. 5. Exploration of optimal kernel and schedule.

가. 최적 스케줄 탐색

최적 스케줄 탐색 단계에서는 쓰레드 수와 각 쓰레드의 작업 할당량을 변화시키면서 성능을 측정하여 최적의 해를 찾아 나간다. 최소 1개 이상의 쓰레드가 존재하여야 연산을 수행할 수 있으며, 전체 코어의 수를 넘어가는 쓰레드의 생성은 성능 저하를 유발하기 때문에 이 경우는 탐색에서 제외한다. 이처럼 쓰레드 수의 최댓값을 한정하더라도, 작업 분배 경우의 수와 곱해지면 총 경우의 수가 여전히 많다. 따라서 탐색 범위를 줄여 실행 시간 내에 최적의 스케줄을 찾는 것이 필요하다.

본 연구에서는 작업 분배에 활용되는 쓰레드 수를 클러스터 내 CPU 코어의 수로 제한한다. 작업 분배의 경우, 동일 클러스터 내의 코어의 성능은 동일하므로, 동일 클러스터 내의 쓰레드는 항상 동일한 작업량을 배분함으로써 탐색할 경우의 수를 줄인다. 예를 들어 Big 코어가 2개, Little 코어가 4개가 탑재된 그림 1의 경우, 다음과 같이 세 가지 조합으로 쓰레드 구성를 구성하고 탐색한다.

- Big 클러스터 쓰레드 2개
- Little 클러스터 쓰레드 4개
- Big 클러스터 쓰레드 2개 + Little 클러스터 쓰레드 4개

본 연구에서는 프로파일 기반으로 생성된 후보 값을 활용하여 최적 쓰레드 수와 최적 작업량 배분을 탐색한다. 그림 6의 알고리즘은 최적 쓰레드 수를 찾기 위하여 모든 경우를 순차적으로 탐색하고, 최적의 작업 배분을 찾기 위해서는 다양한 사용자 정의 탐색 알고리즘(랜덤, 그리드, 베이지안)을 사용한다.

그림 6의 줄 4는 탐색 알고리즘에 따라 임의의 스케줄을 후보로 지정하는 코드이다. 후보 스케줄은 디바이스에서의 실행(줄 5)을 통해 성능이 측정되고, 기존 스

```
clusters := {UC(n, r) where n=number of clusters, 1≤r≤n}
1 : function search_schedule(alg, kernel)
2 :   for cluster in clusters
3 :     while trial<max_trial && has_next_candidate(cluster)
4 :       cand := next_candidate_partition(cluster)
5 :       measure(cand, alg, kernel)
6 :       if best.time > cand.time
7 :         best := cand
8 :       trial++
```

그림 6. 최적 스케줄 탐색 알고리즘
Fig. 6. The algorithm of optimal schedule exploration.

케줄과의 비교(줄 6)를 통해 최적 스케줄을 교체(줄 7)하게 된다. 사용자는 max_trial을 지정함으로써, 무한히 후보를 찾지 않고 특정 시점에 탐색을 종료하게 된다.

나. 최적 커널 구성 탐색

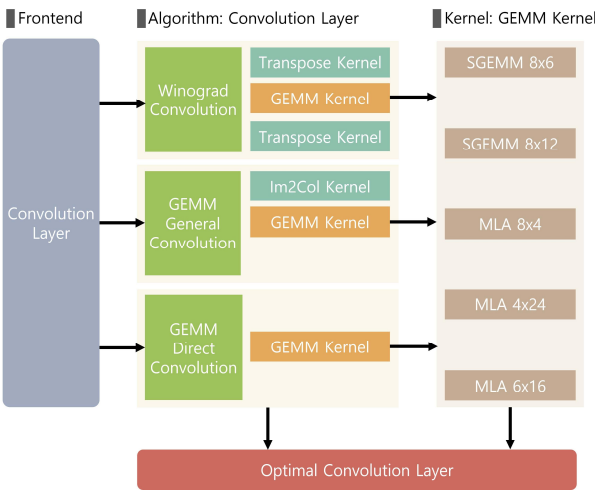

그림 7. 컨볼루션 연산의 최적 커널 탐색의 경우의 수
Fig. 7. The number of kernels for convolution.

최적 커널 구성 탐색 단계에서는 각 연산에 최적인 연산 방법과 커널을 탐색한다. 예를 들어 그림 7과 같은 컨볼루션 연산의 경우, 표 1과 같이 세 가지의 연산 방법이 있고, 각 연산 방법에 표 2와 같이 5가지 이상의 커널을 조합하여 최종적으로 커널을 구성할 수 있다. 따라서, 컨볼루션 연산 당 총 15가지의 경우가 발생하고, 본 논문에서는 모든 경우의 수를 탐색한다.

그림 8은 최적 커널 구성 탐색 알고리즘의 의사 코드를 나타낸다. 줄 2와 3은 총 15가지의 경우의 수를 모두 탐색하는 코드를 나타내며, 줄 4에서 *search_schedule*의 사용자 설정값에 따라 그림 6의 최적 스케줄링 탐색 알고리즘 사용 유무로 분기된다. *search_schedule* 값이 *True* 라면, 커널 구성마다 최적 스케줄링을 추가적

```
algs := ['winograd', 'gemm_general', 'gemm_direct']
kernels := ['sg8x4', 'sg8x12', 'mla8x4', 'mla4x24',
            'mla6x16']
1 : function search_kernels()
2 :   for alg in algs
3 :     for kernel in kernels
4 :       if search_schedule
5 :         cand := search_schedule(alg, kernel)
6 :       else cand := measure(default, alg, kernel)
7 :       if best.time > cand.time
8 :         best := cand
```

그림 8. 최적 커널 구성 탐색 알고리즘
Fig. 8. The algorithm of optimal kernel configuration exploration.

으로 탐색하여 최적의 커널 구성과 스케줄링을 동시에 찾게 된다.

2. 탐색 알고리즘

최적 스케줄 탐색을 위해 모든 경우의 수를 순차적으로 탐색한다면 연산 당 수십 시간이 소요된다. 본 논문에서는 탐색 시간을 최소화하면서 최적의 스케줄을 찾기 위해 샘플링 기반 탐색을 사용하며, 샘플링 방법으로 랜덤 탐색, 그리드 탐색, 베이지안 최적화를 활용한다.

랜덤 탐색은 사전에 정의된 범위 내에서 무작위로 선택하여 최적의 조합을 탐색하는 방식으로 진행된다. 이를 통해, 모든 경우를 시도하지 않고도 최적에 가까운 조합을 찾을 수 있다[4]. 랜덤 탐색은 탐색 대상의 성능 분포가 매우 자연적일 경우, 최소한의 탐색으로도 어느 수준 이상의 해를 찾을 수 있다는 장점이 있다. 그러나 매우 좁은 범위의 해나 궁극적인 해를 찾아야 한다면, 순차 탐색만큼의 샘플링이 필요하다.

그리드 탐색은 탐색 공간을 일정한 간격으로 나누고 하나의 포인트를 선택한다. 각 포인트에서의 평가를 통하여 다음 탐색의 방향을 결정하게 되며, 최적의 해에 점점 더 가까워지도록 탐색하는 방법으로, 구현하기 쉽고 간단하다. 하지만, 차원이 증가할수록 탐색 공간이 기하급수적으로 증가하고, 탐색 대상의 성격에 따라, 그리드 탐색 방식이 잘 적용되지 않을 수도 있다는 단점이 있다.

베이지안 최적화는 목적 함수의 값을 예측하는 모델을 이용하여 최적의 해를 찾아낸다. 이전 탐색 결과를 기반으로 다음 탐색을 수행하기 때문에 그리드 탐색이나 랜덤 탐색보다 효율적으로 탐색할 수 있다. 본 논문에서는 TPE 기반의 오픈소스 라이브러리를 사용하였다[5].

Ⅳ. 실 험

1. 실험 환경

제안하는 최적화 방법의 성능을 검증하기 위해 ACL 22.05 버전을 기반으로 최적화 시스템을 구현하였다. 실험에 사용한 딥 러닝 모델은 AlexNet[18]과 VGG16[19], MobileNetV2[20], GoogLeNet[21]이며 ACL 저장소에서 제공되는 예제 코드를 그대로 사용하였다[3]. 각 모델의 가중치를 다운로드 받기 위해 AlexNet은 Caffe Model Zoo[22]를, 그 외의 모델은 Tensorflow Model Zoo[23]를 이용하였다.

실험에 사용된 Arm big.LITTLE 구조의 CPU를 탑재한 임베디드 보드는 모두 3종으로, Tinker Edge R[6], Odroid N2+[24], Snapdragon 865 Hardware Development Kit(SD 865 HDK)[25]이다. 표 3은 각 보드의 big.LITTLE 구조 정보를 보여준다.

Tinker Edge R은 LITTLE 코어가 4개, big 코어가 2개를 탑재하고 있고, Odroid N2+는 반대로 LITTLE 코어가 2개, big 코어 4개이다. SD 865 HDK는 LITTLE 코어 4개, big 코어 3개, Prime 코어 1개로 3개의 클러스터가 존재한다.

표 3. 실험에서 사용한 임베디드 보드
Table 3. The embedded boards used in the experiment.

		Tinker Edge R	Odroid N2+	SD 865 HDK
SoC 명		RK3399Pro[6]	S922X[24]	Kyro 585[25]
Little	Model	A53	A53	A55
	Core	4	2	4
	Clock	1.4 Ghz	2.0 Ghz	2.0 Ghz
Big	Model	A72	A73	A77
	Core	2	4	3
	Clock	1.8 Ghz	2.4 Ghz	2.4 Ghz
Prime	Model	X	X	A77
	Core			1
	Clock			2.84 Ghz

실험 중 DVFS에 의한 성능변화를 줄이기 위해 CPU 주파수를 고정하였다. 또한, 반복 실행 시 이전 실행에서 적재된 캐시 히트로 인한 성능 향상 효과를 제거하기 위해 캐시 데이터를 매번 리셋하며, 성능 측정 사이마다 충분히 냉각하고, 성능을 12회 측정 후 최대, 최소를 제외한 절사 평균값을 활용하였다.

2. 실험 결과

최적 스케줄 탐색과 최적 커널 구성 탐색 각각의 성능 향상 효과와 두 가지 최적화를 함께 사용하였을 경우의 성능 향상 정도를 측정하기 위해, 3종의 보드에서 4종의 딥 러닝 모델을 대상으로 실험하였다. 각각 그림 6과 그림 8의 알고리즘을 사용하였으며, 최적 스케줄 탐색의 경우 3종의 샘플링 방법을 각각 200회씩 총 600회 탐색하였다.

표 4는 Tinker Edger R에서 각 딥 러닝 모델의 성능 최적화 결과를 나타낸다. 성능은 1회 평균 추론 시간(ms)으로 나타냈고, 괄호 안의 수치는 성능 향상률을 의미한다. Baseline은 ACL을 추가적인 최적화 없이 사용하였을 때의 결과이고, Schedule은 스케줄 탐색 최적화만 적용했을 때의 결과이다. Kernel은 커널 구성 탐색만 적용했을 때의 결과이고, All은 두 가지 최적화를 모두 적용했을 때의 결과이다. 실험 결과, 커널 구성과 스케줄 최적화가 딥 러닝 모델에 따라 주는 영향도는 각각 차이가 나지만, 대체로 높은 성능 향상을 보였고, 두 가지 최적화를 동시에 적용하였을 경우가 각각 적용 효과의 합보다 더 커지는 경우도 발생하여 최대 83%의 성능이 향상되었다. 다만, MobileNetV2의 경우 최적 스케줄 탐색 최적화의 효과가 미미했다.

표 4. Tinker Edge R 보드에서의 실험 결과 (ms)
Table 4. The experimental results on Tinker Edge R (ms).

Opt. \ Model	AlexNet	VGG16	MobileNetV2	GoogLeNet
Baseline	113.0	736.0	88.7	161.1
Schedule	91.1(24%)	497.7(48%)	87.6(1%)	139.8(15%)
Kernel	98.2(15%)	672.3(9%)	76.7(16%)	103.6(56%)
All	82.4(37%)	465.5(58%)	76.9(15%)	88.0(83%)

Odroid N2+에서 기본 ACL을 사용 시, 식 (1)에 의해 쓰레드를 2개 생성하여 연산하게 된다. Odroid N2+는 4개의 big 코어를 갖고 있으므로 Big 클러스터로 모든 쓰레드가 마이그레이션 되더라도, 모든 코어를 활용하지는 못한다. 표 5는 Odroid N2+에서 표 4와 동일한 실험을 진행한 결과를 나타내며, 추가적으로 기본 ACL를 수정하여 항상 쓰레드를 4개 생성하도록 하고 측정한 성능 결과를 4 Thead 행에 나타냈다. 4 Thread의 결과, big 코어의 활용률이 증가하여 최대 53%의 성능이 향상되었고, 본 논문에서 제안하는 스케줄과 커널 구성 최적화를 추가 적용 시 최대 191% 성능이 향상되었다.

표 5. Odroid N2+ 보드에서의 실험 결과 (ms)
Table 5. The experimental results on Odroid N2+ (ms).

Model / Opt.	AlexNet	VGG16	MobileNetV2	GoogLeNet
Baseline	84.9	611.4	61.7	124.8
4 Thread	64.6(31%)	399.5(53%)	48.2(28%)	85.9(45%)
Schedule	59.6(42%)	358.4(71%)	46.4(33%)	77.8(60%)
Kernel	78.7(8%)	544.6(12%)	54.9(12%)	75.9(64%)
All	55.5(53%)	312.3(96%)	37.2(66%)	42.9(191%)

표 6은 SD 865 HDK에서의 실험 결과이다. 다른 보드에서의 실험과 마찬가지로 최대 266%의 성능이 향상됨을 보였다.

표 6. SD 865 HDK 보드에서의 실험 결과 (ms)
Table 6. The experimental results on SD 865 HDK (ms).

Model / Opt.	AlexNet	VGG16	MobileNetV2	GoogLeNet
Baseline	96.3	264.5	173.0	234.8
Schedule	55.0(75%)	189.8(39%)	86.5(100%)	105.3(123%)
Kernel	65.2(48%)	227.1(16%)	127.8(35%)	173.4(35%)
All	37.5(157%)	167.4(58%)	65.6(164%)	64.2(266%)

3종의 최적 스케줄 탐색 방법(그리드 탐색, 랜덤 탐색, 베이지안 탐색)에 따른 성능 차이를 알아보기 위해 각 임베디드 보드에서 200회의 샘플링으로 제한하고 실험하였다.

그림 9-11은 각각 Edge Tinker R, Odroid N2+, SD 865 HDK에서 그림 6의 최적 스케줄 탐색 알고리즘을 수행하면서, 200회의 후보 값을 탐색하면서 최적해의 성능변화를 보여준다.

그 결과 Edge Tinker R에서 MobileNetV2를 최적화하였을 경우를 제외하고 모든 경우에서 일정 횟수의 샘플링 이후에는 모두 성능이 향상됨을 알 수 있다. 또한, 방법에 따른 최적화 정도나 수렴 속도 역시 크게 차이가 나지 않았다. 하지만 대체로 베이지안 최적화 방법이 다른 방법에 비해 결과적으로는 더 높은 성능 향상을 보인다.

V. 결 론

기존 ACL에 이미 고도로 최적화된 커널 코드가 존재하지만, 딥 러닝 연산마다 다른 특성으로 인해 규칙 기반으로 커널과 스케줄을 결정하기에는 최적화에 한계가 있다.

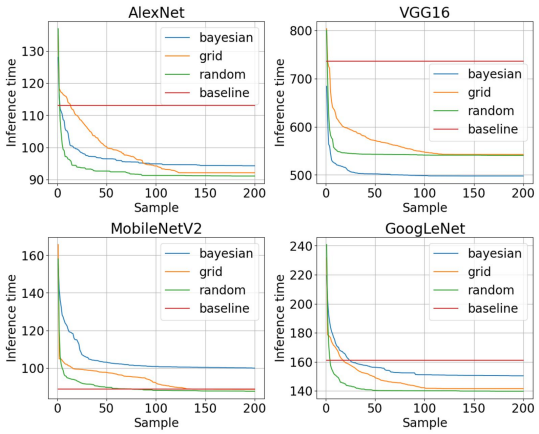

그림 9. Edge Tinker R 보드에서의 샘플링 방법에 따른 성능 변화
Fig. 9. The performance variation according to the sampling method on Edge Tinker R.

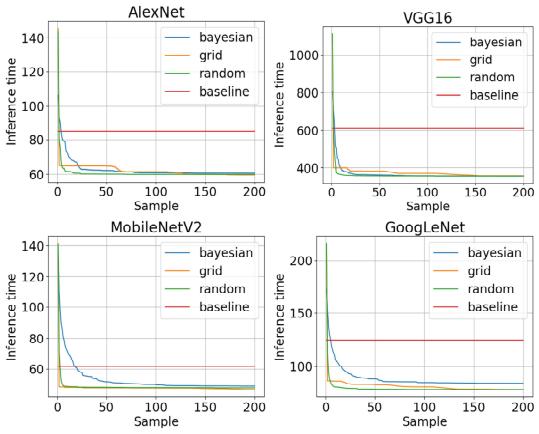

그림 10. Odroid N2+ 보드에서의 샘플링 방법에 따른 성능 변화
Fig. 10. The performance variation according to the sampling method on Odroid N2+.

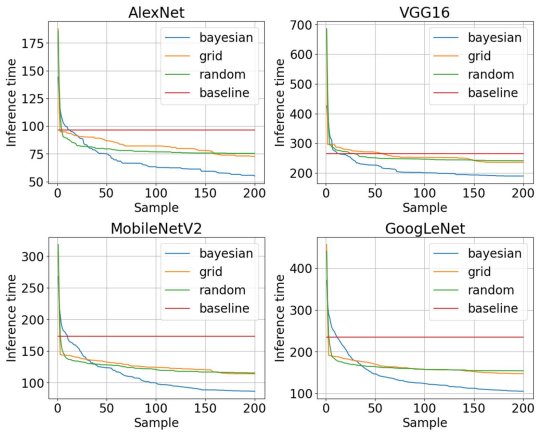

그림 11. SD 865 HDK 보드에서의 샘플링 방법에 따른 성능 변화
Fig. 11. The performance variation according to the sampling method on SD 865 HDK.

본 논문에서는 다양한 Arm big.LITTLE 구조의 CPU 하드웨어에 적용 가능한 프로파일 기반 딥 러닝 연산 최적화 기법을 제시하였다. 제안한 방법의 성능 향상 효과를 검증하기 위해, Edge Tinker R, Odroid N2+, SD 865 HDK 임베디드 보드에서 AlexNet, VGG16, MobilenetV2, GoogLeNet 모델에 대해 실험했고, 최대 266% 성능 향상을 보였다. 또한, 본 성능 향상 수준을 달성하기 위해 200회 미만의 샘플링만으로도 충분함을 입증하였다.

제안하는 알고리즘은 ACL뿐만 아니라, 다양한 딥 러닝 가속 하드웨어 라이브러리에서 적용할 수 있을 것으로 기대되며, 향후 AutoScheduler[26] 등과 같이 컴파일러 자동튜닝 기법과의 성능을 비교하고 개선하는 연구를 진행할 예정이다.

REFERENCES

[1] C. J. Wu et al., "Machine Learning at Facebook: Understanding Inference at the Edge", 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 331-344, Washington DC, USA, 2019

[2] M. Boettcher, B. M. Al-Hashimi, M. Eyole, G. Gabrielli and A. Reid, M. "Advanced SIMD: Extending the reach of contemporary SIMD architectures," 2014 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1-4, Dresden, Germany, 2014

[3] Arm, "Arm Compute Library", https://github.com/ARM-software/ComputeLibrary/tree/v22.05 (accessed Mar. 2023)

[4] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization", The Journal of Machine Learning Research, vol. 13, pp. 281-301, Feb. 2012

[5] J. Bergstra, D. Yamins and D. D. Cox, "Making a science of model search: hyperparameter optimization in hundreds of dimensions for vision architectures", ICML'13: Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28, pp. 115-123, Atlanta GA, USA, Jun. 2013

[6] Rockchip Electronics, "Rockchip RK3399Pro Datasheet", Dec. 2018, https://rockchip.fr/RK3399Pro%20datasheet%20V1.1.pdf (accessed Apr. 2023)

[7] ARM big LITTLE MP tree, https://git.linaro.org/arm/big.LITTLE/mp.git/ (accessed Apr. 2023)

[8] C. Scordino et al., "Energy-aware real-time scheduling in the linux kernel", SAC '18: Proceedings of the 33rd Annual ACM Symposium on Applied Computing, pp. 601-608, Apr. 2018

[9] A. Stevens, "Introduction to AMBA® 4 ACE™ and big. LITTLE™ Processing Technology", ARM White Paper, CoreLink Intelligent System IP by ARM, 2011

[10] M. Abadi et al., "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems", Google Research White Paper, Nov. 2015

[11] ONNX Runtime developers., "ONNX Runtime [Computer software]", 2018, https://github.com/microsoft/onnxruntime, (accessed Apr. 2023)

[12] A. Butko et al., "Position Paper: OpenMP scheduling on ARM big.LITTLE architecture", Conference: International Workshop on Programmability and Architectures for Heterogeneous Multicores MULTIPROG 2016, Prague, Czech Republic, Jan. 2016

[13] A. Anderson, A. Vasudevan, C. Keane and D. Gregg, "Low-memory GEMM-based convolution algorithms for deep neural networks", CoRR, vol. arXiv:1709.03395, pp. 1-13, Sep. 2017

[14] P. Maji 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), pp. 1-5, Feb. 2019

[15] J. Zhang, F. Franchetti and T. M. "Low, High Performance Zero-Memory Overhead Direct Convolutions", International Conference on Machine Learning. PMLR, Jul. 2018.

[16] M. de Prado etl al., "Automated Design Space Exploration for Optimized Deployment of DNN on Arm Cortex-A CPUs," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 40, no. 11, pp. 2293-2305, Nov. 2021

[17] M. Dukhan, "The Indirect Convolution Algorithm", CoRR, vol. arXiv:1907.02129, pp. 1-10, Jul. 2019

[18] A. Krizhevsky, I. Sutskever and G. E.

Hinton, "Imagenet classification with deep convolutional neural networks", Communications of the ACM, vol. 60, pp. 84-90, May. 2017
[19] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition", 3rd International Conference on Learning Representations (ICLR 2015), pp. 1-14, Apr. 2015
[20] M. Sandler et al., "MobileNetV2: Inverted Residuals and Linear Bottlenecks", 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 4510-4520, Jan. 2018
[21] C. Szegedy et al. "Going deeper with convolutions", 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 1-9, Los Alamitos, CA, USA, Jun. 2015
[22] Y. Jia et al., “Caffe: Convolutional architecture for fast feature embedding", Proceedings of the 22nd ACM international conference on Multimedia, pp. 675-678, Jun. 2014
[23] H. Yu et al., "TensorFlow Model Garden", 2020
[24] ODROID Wiki, "ODROID-N2/N2Plus", https://wiki.odroid.com/odroid-n2/odroid-n2 (accessed Apr. 2023)
[25] WikiChip, "Snapdragon 865 - Qualcomm", https://en.wikichip.org/wiki/qualcomm/snapdragon_800/865 (accessed Apr. 2023)
[26] Lianmin Zheng et al., “Ansor: Generating highperformance tensor programs for deep learning”, 14th USENIX Symposium on Operating Systems Design and Implementation, Nov. 2020.

저 자 소 개

차 주 형(비회원)
현재 동의대학교 응용소프트웨어 공학과 학사과정 재학.
<주관심분야 : 이기종 컴퓨팅, 임베디드, 최적화, 인공지능>

이 제 민(비회원)
2011년 충남대학교 컴퓨터공학과 학사 졸업
2017년 충남대학교 컴퓨터공학과 박사 졸업
2017~2018년 한국과학기술원 박사후연구원
현재 한국전자통신연구원 선임연구원
<주관심분야 : 인공지능 모델 경량화, 컴파일러, 모바일 컴퓨팅>

권 용 인(비회원)
2008년 한국과학기술원 전기 및 전자공학과 학사 졸업.
2010년 서울대학교 전기컴퓨터 공학부 석사 졸업.
2015년 서울대학교 전기컴퓨터 공학부 박사 졸업.
현재 한국전자통신연구원 선임연구원
<주관심분야 : 인공지능, 반도체, 컴파일러, 최적화>