인공지능 프로세서 컴파일러 개발 동향

Trends of Compiler Development for AI Processor

김진규 (J.K. Kim, kimjk@etri.re.kr) 인공지능프로세서연구실 책임연구원
김혜지 (H.J. Kim, hyejikim@etri.re.kr) 인공지능프로세서연구실 연구원
조용철 (Y.C.P. Cho, cho@etri.re.kr) 인공지능프로세서연구실 선임연구원
김현미 (H.M. Kim, chaos0218@etri.re.kr) 인공지능프로세서연구실 선임기술원
여준기 (C.G. Lyuh, cglyuh@etri.re.kr) 인공지능프로세서연구실 책임연구원
한진호 (J. Han, soc@etri.re.kr) 인공지능프로세서연구실 책임연구원/실장
권영수 (Y. Kwon, yskwon@etri.re.kr) 지능형반도체연구본부 책임연구원/본부장

ABSTRACT

The rapid growth of deep-learning applications has invoked the R&D of artificial intelligence (AI) processors. A dedicated software framework such as a compiler and runtime APIs is required to achieve maximum processor performance. There are various compilers and frameworks for AI training and inference. In this study, we present the features and characteristics of AI compilers, training frameworks, and inference engines. In addition, we focus on the internals of compiler frameworks, which are based on either basic linear algebra subprograms or intermediate representation. For an in-depth insight, we present the compiler infrastructure, internal components, and operation flow of ETRI's "AI-Ware." The software framework's significant role is evidenced from the optimized neural processing unit code produced by the compiler after various optimization passes, such as scheduling, architecture-considering optimization, schedule selection, and power optimization. We conclude the study with thoughts about the future of state-of-the-art AI compilers.

KEYWORDS AI Processor, AI Compiler, Deep Learning Framework

Ⅰ. 서론

최근 들어 딥러닝을 활용한 인공지능 기술이 발전함에 따라 의료, 금융, 제조, 보안, 운송, 교육 등의 다양한 분야에서 널리 활용되는 추세이다.

딥러닝 기술은 1998년에 뉴욕대학교 Yann LeCun 교수가 MNIST 데이터셋을 이용하여 필기체 숫자 인식이 가능한 CNN 기반 LeNet을 처음으로 제안하고[1], 2012년도에 Alex Krizevsky가 ImageNet 사물 인식 경진대회에서 AlexNet을 제안하여 우승한 이

* DOI: https://doi.org/10.22648/ETRI.2021.J.360204
* This work was supported by the ICT R&D program of MSIT/IITP[2018-0-00195, Artificial Intelligence Processor Research Laboratory].

래[2] 사람보다 더 뛰어난 시각 인지 성능을 보이는 2015년 ResNet-152(top-5 검출률 96.43%), 2021년 EfficientNet-L2(top-5 검출률 98.8%)에 이르기까지 매년 계속해서 발표되었다[3,4]. 그리고 자연어 기반 음성 인식 및 상호나 자동차 번호판 등의 문자 인식에 널리 활용되는 시퀀스 정보 기반 인지를 위한 RNN(Recurrent Neural Network) 및 LSTM (Long Short-Term Memory) 인공지능 알고리즘도 계속 발표되고 있다. 이러한 인공지능 응용 서비스를 위한 하드웨어의 성능의 요구 사양은 나날이 높아지는 추세이다.

대규모 뉴럴 네트워크를 처리하기 위한 인공지능 프로세서의 개발이 전 세계적으로 국가별 또는 회사별로 매우 경쟁이 치열해지는 양상이다. 인공지능 프로세서 개발은 NVIDIA가 V100, A100 등 강력한 성능을 갖는 GPU 칩을 계속 출시하면서 반도체 칩 계산 성능 측면에서 선도하고 있으며, MLPerf의 벤치마크 결과를 보면 학습을 위한 고성능 컴퓨팅(HPC) 및 추론용 데이터센터의 가속기로 활용되고 있다[5]. 테슬라는 완전자율주행이 가능하도록 브로드컴과 함께 고성능 컴퓨팅이 가능한 하드웨어 4.0을 준비 중이다. 이외에도 구글, 애플, 페이스북 및 아마존에서도 인공지능 반도체 개발에 나서고 있으며, 중국도 대규모로 인공지능 반도체 개발 투자에 나서며 화웨이의 Kirin 칩 등 연산 성능을 크게 향상시키는 방향으로 개발하는 추세이다.

초기의 인공지능 전용 프로세서 개발 경향은 학습보다는 주로 추론의 성능 가속에 초점을 맞추어 진행되었다. 이는 학습은 데이터 센터 등의 고성능 서버에서 수행하고 실제 인공지능 서비스 시에는 추론만 사용한다는 가정이 있었기 때문이다. 그러나 딥러닝 알고리즘 기술이 발전함에 따라 추론뿐만 아니라 학습도 병행 가능한 인공지능 반도체의 수요가 많아지고 있다. 센서노드나 엣지노드에서 실시간 학습이 가능하면 데이터 전송에 필요한 네트워크 대역폭을 낮출 수 있고, 디바이스별 고유 학습이 가능하기 때문이다.

대규모 학습과 추론 처리 성능을 극대화하기 위해서는 인공지능 프로세서의 자체 연산 성능도 중요하지만, 이를 운용하기 위한 인공지능 컴파일러의 최적화도 함께 요구된다.

본 고에서는 먼저 Ⅱ장에서 다양한 인공지능 컴파일러의 종류 및 개발 동향을 살펴본다. 그리고 Ⅲ, Ⅳ장에서는 BLAS 기반 하드웨어 가속과 IR 기반 하드웨어 가속 방식에 관해 살펴보고, Ⅴ장에서는 ETRI의 개발 예를 소개하고, Ⅵ장에서 결론을 맺도록 한다.

Ⅱ. 컴파일러 개발 동향

인공지능 컴파일러는 유향 비순환 그래프(DAG: Directed Acyclic Graph)로 표현된 CNN, DNN 등의 뉴럴 네트워크를 하드웨어 명령어 집합(Instruction Set)으로 변환하는 일련의 과정을 의미한다. 이 과정에서 그래프의 최적화, 하드웨어별 자원 할당 및 분배과정이 동반된다.

하드웨어 의존성이 강하기 때문에 인공지능 반도체 개발과 병행되어 개발된다.

1. Arm NN/ArmCL

Arm NN SDK는 TensorFlow, PyTorch 등의 다양한 딥러닝 프레임워크 모델을 입력으로 사용할 수 있다. Arm Cortex-M/A, Arm Mali GPU 그리고 Arm NPU인 Ethos-N AI 프로세서에서 동작 가능한 명령어로 변환하는 과정을 수행한다. 그림 1에 이와 관련한 Arm ML(Machine Learning) 구조를 보

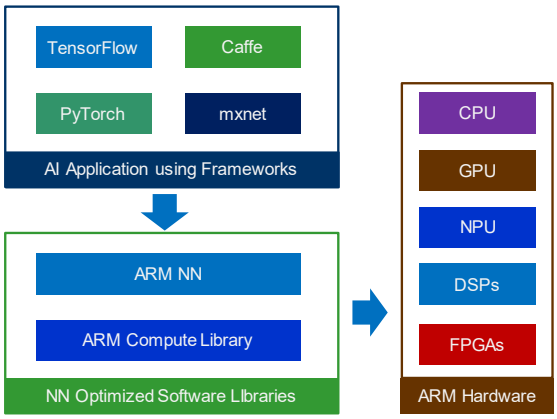


그림 1 ARM ML 컴퓨팅 플랫폼 구조[6]

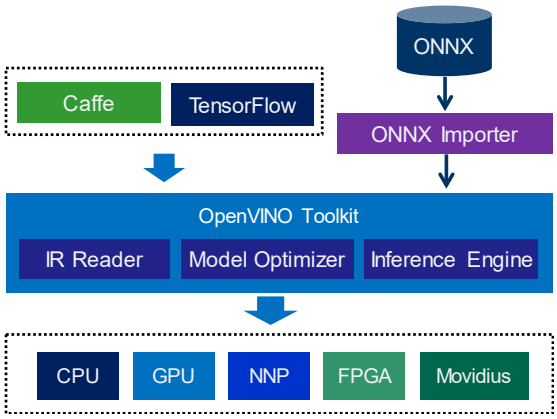


그림 2 Intel OpenVINO 구조[9]

인다.

Arm NN SDK는 Arm CL(Compute Library) 내에 뉴럴 네트워크 연산자별로 구현되어 있는 API 함수를 사용하여 컴파일 과정을 수행한다. Arm CL은 Arm에서 개발한 컴퓨터 비전 및 수치 연산 라이브러리로서, 프로세서 내부의 SIMD(Single Instruction Multiple Data) 가속 엔진인 NEON 및 SVE(Scalable Vector Extension)로 구현되거나 Mali GPU를 사용하기 위해 OpenCL(Open Computing Language) 커널로 구현된 API 함수로 동작한다[6,7].

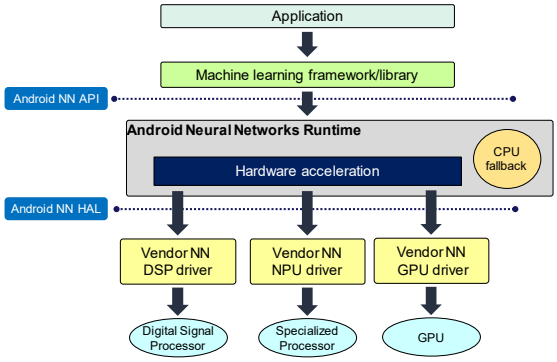
2. Intel OpenVINO/nGraph

인텔은 Atom, Xeon 등의 CPU, Intel Integrated Graphics와 같은 GPU, USB 인터페이스를 사용한 Movidius VPU나 MyraidX VPU 등에서 ONNX(Open Neural Network eXchange)로 표현된 뉴럴 네트워크 프로세싱이 가능하도록 OpenVINO 툴킷을 제공하고 있다[8,9]. ONNX는 뉴럴 네트워크 간의 상호 변환 및 백앤드(Backend) 하드웨어 처리를 위한 오픈 표준이다. 그림 2에 나타난 것과 같이 ONNX 레이어(Layer)별 연산자는 상위수준(High Level) IR로 변환된 후에 일대일로 대응되는 nGraph IR로 매핑되어 추론 엔진에서 수행 가능한 하위 수준의 IR로 변환되어 수행된다. 딥러닝 프레임워크의 종류가 매우 다양하기 때문에 TensorFlow처럼 직접적으로 연산자가 매핑되어 지원될 수도 있으며, PyTorch와 같이 파이썬 클래스(nn.Module) 뉴럴 네트워크 모델을 ONNX 형식으로 변환한 후에 처리하는 간접적인 방식도 지원된다.

3. Android Neural Networks

Android NN(Neural Networks) API는 모바일 단말기에서 기계학습을 위해 개발된 C/C++ API 라이브러리이다. 뉴럴 네트워크를 빌드하고 학습시키는 높은 수준의 머신러닝 프레임워크에 필요한 기능 레이어를 제공하도록 설계되어 있다.

그림 3에 나타난 것처럼 NN API를 사용하여 Android NN Runtime을 통해 실행할 수 있으며, 하드웨어 가속은 DSP, NPU, GPU가 지원된다. 지원되지 않는 뉴럴 네트워크 연산자의 경우에는 CPU



출처 Android, https://developer.android.com/ndk/guides/neuralnetworks

그림 3 Android NNAPI 구조[10]

fallback을 이용하여 수행할 수 있다[10].

4. AMD ROCm

AMD에서 발표한 ROCm(Radeon Open Compute stack) 소프트웨어 툴셋 내부에 포함된 MIOpen은 딥러닝용 어플리케이션에서 주로 사용되는 연산자(Operator) 라이브러리 형태로 구성되며 GPU를 위한 이종 컴퓨팅 인터페이스 HIP(Heterogeneous-computing Interface for Portability) 및 OpenCL 언어로 작성되어 있다.

HIP는 AMD GPU를 위한 Runtime API 함수 및 커널 언어를 지원하며, NVIDIA의 CUDA 코드를 HIP으로 변환하는 기능도 포함하고 있다. 그래프 최적화 엔진인 MIGraphX를 통해 ONNX 그래프가 해석되어 ONNX 내부의 연산자로 할당됨으로써 ONNXRuntime에서도 동작이 가능하다[11].

5. NVIDIA TensorRT

TensorRT는 딥러닝 추론 성능을 극대화하기 위해 NVIDIA에서 개발한 소프트웨어 개발 도구이다. 텐서 및 레이어 간 결합(Fusing)을 통한 최적화, 커널 튜닝(Kernel tuning), 텐서(Tensor) 메모리의 동적 운용 및 8-bit integer나 8-bit float-point 데이터 형식 지원 등을 통해 추론 동작의 처리율을 높일 수 있다[12].

6. Xilinx Vitis AI

Vitis AI 개발 도구는 Alveo 가속기와 같은 Xilinx 하드웨어 플랫폼에서 인공지능 추론 동작의 가속을 위한 소프트웨어 툴킷이다. Caffe, PyTorch, TensorFlow 모델을 지원하며 하드웨어 가속 엔진은 DPU(Deep Learning Processing Unit)를 사용한다. Vitis AI 개발 도구 내에 AI Compiler, AI Quantizer, AI Optimizer가 포함되어 있으며, XRT(Xilinx Runtime) 라이브러리와 함께 AI Profiler를 사용할 수 있다[13].

III. DNN/BLAS 기반

인공지능 학습을 위한 딥러닝 프레임워크의 학습 처리 능력을 좌우하는 가장 큰 요소는 DNN 라이브러리와 고속 BLAS의 지원 여부이다. 학습 시에 사용하는 CPU 하드웨어가 Intel x86 계열인지, ARM 계열인지 및 GPU 하드웨어가 NVIDIA 혹은 AMD 계열인지에 따라서 DNN 및 BLAS 라이브러리가 다르다. 따라서 학습 프레임워크 내 레이어별 연산자들이 DNN 및 BLAS API 함수와 어떠한 방식으로 연결되는지에 따라 딥러닝 프레임워크의 추론이나 학습 성능이 크게 다를 수 있다.

개발자들이 널리 사용하는 TensorFlow와 PyTorch 등에서 사용하는 BLAS를 정리하면 그림 4와 같이 표현된다[14,15].

오픈 소스로 개발된 딥러닝 프레임워크는 다차

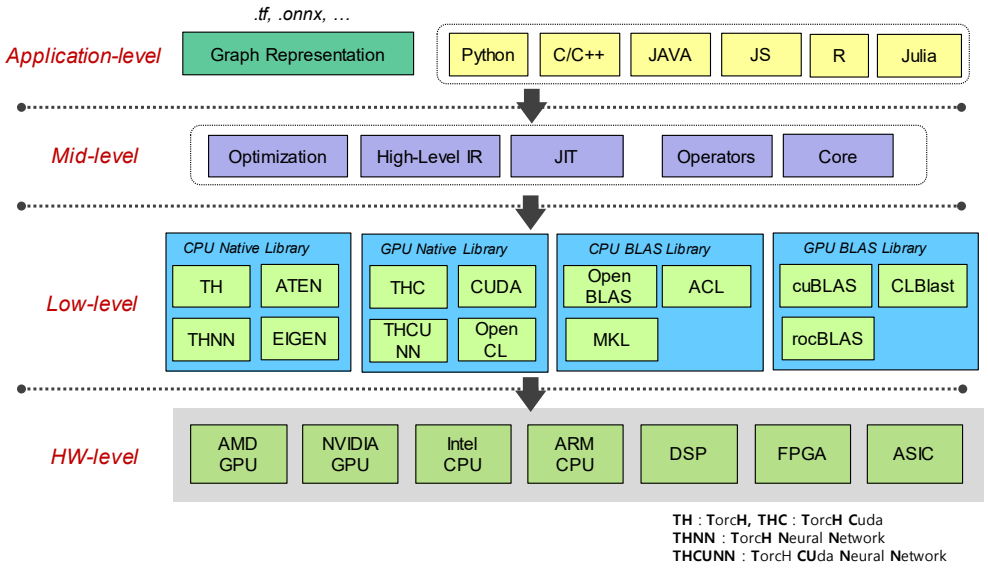


그림 4 BLAS와 SW 스텍[14,15]

원 데이터인 텐서(Tensor) 기반의 DNN 라이브러리와 행렬 기반 수학 연산을 지원하는 BLAS를 주로 사용하고 있다. BLAS API 함수들은 고속 처리를 위해 CPU의 경우에는 주로 어셈블리 언어로 구현되어 있으며, GPU의 경우에는 자체 언어(NVIDIA의 경우에는 CUDA), 내장 함수 또는 OpenCL 언어로 구현되어 있다.

BLAS는 딥러닝 프레임워크를 이용한 뉴럴 네트워크 학습에서 반드시 필요한 수학 연산 라이브러리이고 하드웨어 의존성이 크기 때문에 학습 속도 향상을 고려한다면 BLAS 사용을 배제하기는 쉽지 않다. 그러나 딥러닝 학습 과정이 종료된 후, 배포(Deploy)용 추론 모델만 사용한다고 가정할 경우, 컴파일러를 사용한 최적화 과정에서 레이어 결합(Fusing) 등을 포함한 연산 흐름(Operation Flow) 최적화가 중요하기 때문에 BLAS 사용이 크게 요구되지는 않는다. 예를 들어, 마이크로소프트에서 구현한 ONNXRuntime은 다양한 하드웨어를 지원하면서 추론과정에 대한 최적화가 잘 구현되어 있다[16,17].

BLAS API 함수 중에서 가장 많이 사용되는 GEMM(GEneral Matrix Multiplication) 함수는 fully-connected 레이어에서는 일대일로 대응되지만 convolutional 레이어의 경우에는 GEMM API 함수를 적용하기 전에 im2col(image to column) 함수를 이용하여 입력 텐서 데이터를 행렬에 맞게 전개해 주어야 한다. ONNXRuntime의 경우에는 BLAS를 사용하는 대신 활성화(Activation) 함수와 결합된 CONV 레이어 연산자를 어셈블리로 구현함으로써 추론 속도 향상화를 꾀하고 있다. 이외에도 NVIDIA의 cuDNN 라이브러리 등과 같이 뉴럴 네트워크 레이어별 연산자를 별도로 구현하여 가속하거나 학습 시에 필요한 순방향 처리(Forward Processing)나 역방향 처리(Backward Processing) 동작을 최적화하기도 한다.

1. CPU 기반

CPU 기반 BLAS로는 ATLAS, OpenBLAS, MKL, Arm PL(Performance Library) 등이 존재하며, 딥러닝

프레임워크에서 많이 사용되는 Eigen이나 Aten 등도 넓은 의미에서 BLAS에 포함될 수 있다[18-20].

BLAS 함수의 종류는 입력 데이터의 조합이 원소(Element), 벡터(Vector) 혹은 행렬(Matrix)인지에 따라 3단계의 레벨(Level)로 구성된다. CPU의 BLAS API는 함수명부터 입력 인자(Argument)의 순서까지 동일하기 때문에 BLAS 라이브러리를 서로 교차하여 사용하기가 용이하다.

2. GPU 기반

GPU 기반 BLAS로는 개발자들이 가장 많이 사용하는 NVIDIA의 cuBLAS가 가장 유명하며, ROCm 플랫폼에서 동작하는 AMD에서 개발한 rocBLAS도 있다. 이외에도 OpenCL로 구현된 clBLAS, clDNN과 OpenCL BLAS의 성능을 높이기 위해 하드웨어에 적합하도록 튜닝이 가능한 CLBlast 등이 있고, C++ 단일 소스로 구현 가능한 SYCL 기반의 SYCL-BLAS나 SYCL-DNN 등도 존재한다[21-23].

3. BLAS 성능

BLAS 함수의 성능은 하드웨어 리소스에 행렬의 데이터를 어떻게 잘 배분하는가에 따라 결정된다. 하드웨어의 가속기의 최대 연산 처리 성능이 높다고 해도 API 함수로 입력된 행렬 데이터가 하드웨어 자원에 적절하게 배분되지 않는다면 주어진 하드웨어 성능을 끌어낼 수가 없기 때문에 효율성에서 매우 떨어진다.

인공지능에서 가장 널리 사용되는 GEMM API는 행렬 곱셈인데, 행과 열의 크기가 수만에서 수십만 이상에 이를 정도로 매우 크기 때문에 이를 작은 행렬로 계속 분해하는 과정이 필요하다.

NVIDIA의 cuBLAS의 경우에는 내부 구현을 공개하고 있지 않지만, CUTLASS[24]를 제공함으로써 개발자에게 쓰레드(Thread), 와프(Warp), 블록(Block), 디바이스(Device) 레벨에서 효율적인 CUDA 코딩 방법을 제시하고 있다. 이를 활용하면 개발자는 GPU 메모리 할당에서 GPU 코어 연산기까지 고려하여 효과적인 CUDA 코딩이 가능하기 때문에 고속 성능을 기대할 수 있다.

Ⅳ. 중간표현(IR) 기반

최근 들어 상위 수준(High Level)의 IR을 사용하여 뉴럴 네트워크의 그래프를 처리하는 방법이 새롭게 떠오르고 있다.

뉴럴 네트워크의 그래프 복잡도가 계속 증가하는 추세이기 때문에 LLVM IR과 같은 하위 수준(Low Level)에서의 최적화 방법뿐만 아니라 상위 수준에서의 최적화 효과도 기대할 수 있다. 또한 다른 딥러닝 프레임워크와의 상호 간 그래프 변환이나 연산자 대응 등을 이용한 이식성이 용이해진다.

1. TensorFlow XLA

TensorFlow 딥러닝 프레임워크의 오픈 소스에 구현되어 있는 XLA(Accelerated Linear Algebra)는 그래프 기반의 뉴럴 네트워크 모델을 가속하는 기능을 수행한다[25]. XLA 프로그래밍 언어는 HLO(High Level Operations) IR이다. XLA와 더불어 BLAS tf.operator를 포함한 전체 TensorFlow의 구조가 그림 5에 표현되어 있다.

HLO는 명령어(Instruction) 기반의 고수준 표현 방식이며, 뉴럴 네트워크의 레이어별로 구성되어 있는 연산자(Operator)와 상호 대응(Mapping)되는 관계를 갖는다. 예를 들어, HLO 명령어 중에는

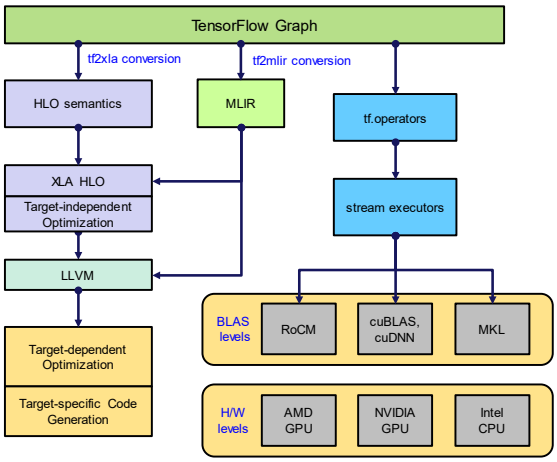


그림 5 TensorFlow 동작 구조[14,25]

'Conv', 'BatchNorm', 'Pad', 'Transpose' 등과 같이 다차원 텐서 기반의 데이터를 처리하기 위한 명령어들이 존재한다.

HLO로 표현된 뉴럴 네트워크는 XLA 내부에 구현된 Compiler(), Executable() 및 TransferManager() 등의 클래스(Class) 함수를 이용하여 실행된다. 주요 기능을 살펴보면, Compiler()는 HLO를 하드웨어에 맞게 기계어로 변환하는 기능을 담당하고, Executable()은 런타임(Runtime) 환경을 구성해 주며, TransferManager()는 호스트(Host)와 디바이스(Device) 간의 통신 기능을 수행한다.

XLA의 가장 큰 장점은 HLO에서의 그래프 최적화를 이용한 가속이 가능하다는 것과 LLVM IR로 변환된 이후, LLVM 자체의 최적화 기능도 기대할 수 있다는 것이다.

TensorFlow는 HLO와 더불어 텐서플로우 그래프에서 MLIR(Multi Level Intermediate Representation)로 변환되어 LLVM IR로 진행하는 방법도 제안하고 있다. 이를 위해 구글 개발자들이 LLVM 프로젝트에 MLIR 관련 서브 프로젝트를 만들어 활동하고 있다[26]. MLIR의 목적은 C, C++, JAVA, CUDA, Swift, Julia 등과 같은 다양한 프로그래밍 언어를 딥러닝 프레임워크에서 지원하고자 할 때, 언어마다 각각의 IR을 구현하여 LLVM IR로 변환하는 과정을 구현해야 하기 때문에 이에 따른 IR의 수가 매우 많아지는 것을 단일 IR로 통합하고자 하는 출발점에서 시작되었다.

IR은 기본적으로 SSA(Static Single Assignment) 기반으로 표현되고, 또한 MLIR의 'dialect' 접두어(Prefix)에 따라 광범위하게 표현이 가능하기 때문에 컴파일러의 백앤드 구현의 다양성을 포함할 수 있다. IR이 추구하는 연산 최적화 과정 또한 MLIR로의 통합이 가능해지기 때문에 컴파일러 구조(Compiler infrastructure)가 간결해지고 컴파일러 구현 비용이 절감되는 효과를 갖는다.

2. ONNX-MLIR

ONNX 기반으로 상위 수준의 IR을 생성하는 프로젝트로 ONNX-MLIR이 있다[27,28].

ONNX-MLIR의 목적은 TensorFlow와 마찬가지로 상위 수준의 IR 명령어를 이용하여 뉴럴 네트워크를 표현함으로써 최적화 및 하드웨어 의존도를 갖는 백앤드 처리의 자유도를 주는 것이다.

먼저 ONNX 그래프가 입력되면 ONNX 연산자로 표현되는 ONNX IR을 생성한다. ONNX IR은 TensorFlow의 HLO와 같은 연산자 단위의 상위 수준 IR이다. ONNX IR은 다음 과정에서 MLIR로 변환되는데, MLIR의 'std', 'llvm', 'affine' 등의 'dialect' prefix를 이용하여 표현된다. MLIR로 표현된 코드는 LLVM 도구를 이용하여 LLVM IR로 변환될 수 있고, 하드웨어 백앤드에 따라 기계어로 출력이 가능하다. ONNX 그래프가 main_func() 함수로 매핑된 공유 라이브러리(Shared Library) 형태로 출력되면 해당 바이너리는 런타임을 이용하여 수행이 가능하다. 그림 6에서는 MNIST ONNX 그래프

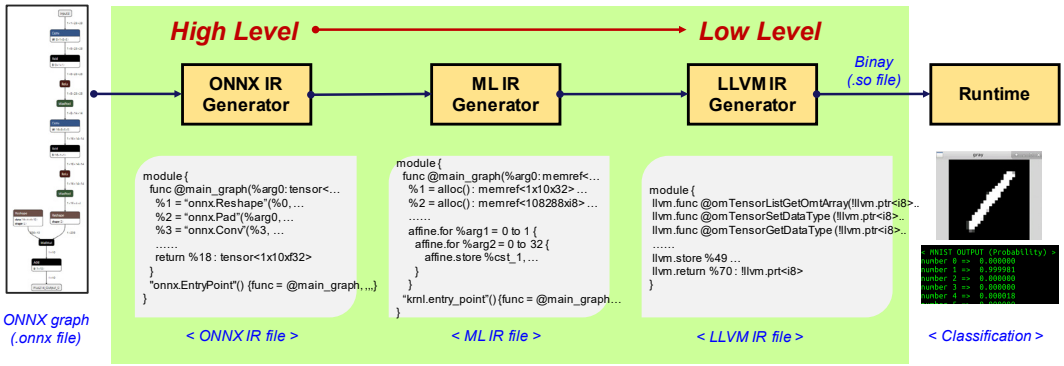


그림 6 ONNX-MLIR 처리 흐름[27,28]

가 ONNX IR, ML IR 그리고 LLVM IR을 거쳐 공유 라이브러리인 mnist.so 파일로 변환된 후, 실행되는 과정을 보인다. main_graph()를 호출하여 수행하는 런타임, MNIST 입력에 대한 이미지 전처리(Pre-processing), 확률값을 출력하기 위한 후처리(Post-processing) 과정을 수행하면 그림 6과 같이 최종 인식 결과를 출력한다.

V. ETRI NPU 컴파일러 환경

2020년에 ETRI에서 제작한 AB9 인공지능 프로세서[29-33]의 넓은 활용을 위해서는 ONNX 프레임워크와의 연계가 중요하다.

ONNX 표현 방법은 TensorFlow나 PyTorch와 같이 널리 사용하는 딥러닝 프레임워크와 연산자 호환성이 뛰어나다. 또한 학습이 완료된 뉴럴 네트워크의 추론 성능을 고도화하기 위한 최적화도 가능하다. 이에 따라 CPU, GPU, NPU과 같이 인공지능 프로세서를 개발하는 글로벌 반도체 설계 업체들은 ONNX 그래프 지원을 추구하고 있다.

AB9을 활용한 ONNX 처리 구조 및 구성이 그림 7에 나타나 있다. 전체 구성은 자체 개발한 AI-

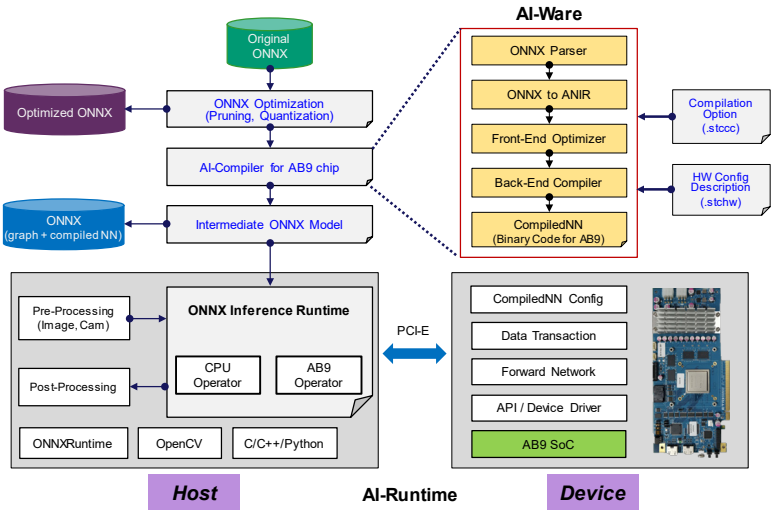


그림 7 ETRI AB9을 활용한 ONNX 처리 구조

Opt 및 AI-Ware, 실행 바이너리를 포함한 AB9 인공지능 프로세서 전용 연산자(AB9 Operator), 그리고 인공지능 프로세서와의 데이터 교환 및 알고리즘 수행을 위한 AB9 런타임으로 구성된다.

1. AI-Opt

AI-Opt는 ONNX 모델을 AB9에서 연산 효율적으로 실행하기 위한 뉴럴 네트워크 최적화 과정이다. 이는 검출 성능을 유지하면서도 추론 속도와 전력 효율을 높이기 위해 사용된다. 학습된 ONNX 모델과 달성하고자 하는 처리 성능을 최적화의 조건으로 제시하면 경량화된 ONNX 모델이 생성되며, 결과물은 AB9에서 실행하기 위한 컴파일러의 입력으로 활용된다.

뉴럴 네트워크 최적화는 대표적으로 희소화 과정과 양자화 과정이 수행된다. 희소화 과정은 검출 성능에 영향력이 적은 불필요한 학습 파라미터를 제거하여 전체 메모리 사용량과 연산량을 줄임으로써 추론 속도를 높이기 위한 것이다. 양자화 과정은 학습 파라미터의 데이터 타입을 8-bit integer나 8-bit float-point로 변환하여 처리함으로써 메모리 사용량을 줄이고 처리 속도를 높임과 동시에 전력 사용량은 감소시킬 수 있다.

2. AI-Ware

AI-Ware는 ONNX 파일의 구문 해석기(Parser)와 AI-컴파일러로 구성된다. 입력된 ONNX 그래프를 연산자별로 파라미터를 분석하여 뉴럴 네트워크 그래프 IR을 표현하기 위해 개발된 ANIR로 변환한다. 변환 과정에서는 컨볼루션(Convolutional), 배치 정규화(Batch Normalization), 활성화(Activation) 및 max-pool 함수 연산자 등을 일괄 처리할 수 있도록 최적화 과정이 수반된다. ANIR로 변환되면 AB9에서 처리 가능한 명령어 집합(Instruction Set)으로 컴파일 과정을 수행한다. 컴파일 과정에서는 입력 데이터를 그룹화시켜 여러 개의 부분(Tiling Part) 데이터로 분할하고 AB9 내부의 시스톨릭 어레이(SA: Systolic Array) 연산기에 적절하게 배분되도록 하는 과정이 수반된다. AB9 내부에 입출력 버퍼(in/out buffer)와 STC(Systolic Tensor Core) 어레이 간에 데이터 이동 및 연산기 할당, 그리고 이에 따른 흐름 제어(Flow control) 등의 동작을 수행한다[29-31].

3. AB9 Operator

ONNXRuntime은 마이크로소프트에서 주도하고 있는 ONNX 그래프를 처리하기 위한 추론 전용 오픈 소프트웨어 툴킷이며, 현재는 추론뿐만 아니라 학습까지 적용시키는 개발을 진행 중이다. ONNXRuntime의 내부 구조는 ONNX 그래프 및 연산자, 하드웨어별로 ONNX 연산자를 처리하기 위한 연산 집행자(Execution Provider), 연산 집행자 내에 존재하는 단위 연산자(Operator)로 구성되어 있으며, 이러한 계층 구조에서 새로운 연산 집행자 및 단위 연산자를 추가할 수 있는 방법도 제시하고 있다.

이에 따라 ETRI에서도 ONNX 그래프를 입력으로 AB9 인공지능 프로세서에서 수행이 되도록 ONNXRuntime 프레임워크를 활용한 개발이 진행 중이다. 이를 위해 ONNXRuntime 시에 수행 가능하도록 실시간으로 실행 바이너리를 생성할 수 있는 Just-In-Time(JIT), STC 컴파일러를 포함한 AI-Ware를 개발하고 있다. AI-Ware를 통해 생성한 AB9 인공지능 프로세서 실행 바이너리는 ONNXRuntime에서 수행하는 전용 연산자(Custom

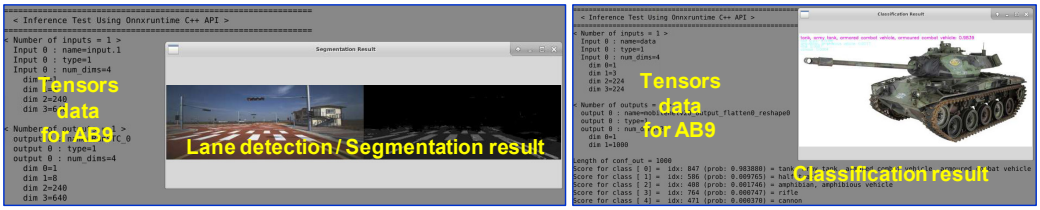


그림 8 AB9을 활용한 ONNX 런타임 수행 예

Operator) 내부의 정보로 입력된다.

전용 연산자는 기존 ONNX 연산자와의 원활한 인터페이스를 위해 적절한 입력/출력 텐서 클래스로 정의되며, 이를 위한 텐서 속성 정보 등이 추가된다. 또한 연산자 처리는 실제 AI-Runtime 등을 통해 입력과 출력 메모리 공간을 할당하고 PCIe 인터페이스를 통해 입력 데이터를 AB9으로 전달하고 실제 연산을 인공지능 프로세서에서 처리한 후 출력 데이터를 다시 받아서 ONNXRuntime 프레임워크로 전달하는 과정을 반복적으로 수행하게 된다.

4. 동작 및 검증

동작 검증 과정은 객체 검출(Object detection), 이미지 분류(Image classification), 이미지 분할(Image segmentation) 등의 응용을 중심으로 ONNX 그래프 기반으로 검증을 진행 중이다.

AB9에서 수행하는 일련의 복잡한 과정이 딥러닝 프레임워크에서는 단일 연산자의 기능으로 보여야 하고, 런타임 과정에서 ONNX 그래프 처리가 원활하게 되려면 입출력 데이터 전달, 메모리 할당, 예외 처리 등에 대한 고려가 많이 필요하다.

그림 8에 ONNX 그래프를 입력으로 이미지 분할 및 이미지 분류 동작을 보인다. 이미지 분류 동작은 소프트맥스(Softmax) 처리를 제외한 레이어들이 AB9 연산자로 수행되고, 이미지 분할 처리 동작은 argmax() 처리를 제외한 레이어들이 AB9 연산자로 처리된다.

VI. 결론

본 고에서는 전 세계적으로 널리 개발되고 있는 인공지능 프로세서와 그의 컴파일러 동향에 관해 살펴보았으며, 컴파일러의 내부를 구성하는 명령어 기반 IR 및 DNN/BLAS 기반 방법 등에 관해 기술하였다.

마지막으로는 ETRI에서 수행하고 있는 AB9 전용 연산자를 활용한 ONNX 처리 과정 및 검증 환경을 제시하였다.

용어해설

ONNX TensorFlow나 PyTorch와 같이 다양한 딥러닝 프레임워크 간에 상호 변환을 위해 MicroSoft 등의 회사들이 참여하여 정의한 뉴럴 네트워크를 표현하기 위한 공개 표준

ONNXRuntime MicroSoft에서 주도적으로 개발하고 있는 ONNX 그래프의 추론 가속화를 위한 프레임워크이며, ARM, NVIDIA, Intel, Rockchip, Xilinx 등의 딥러닝 가속 반도체 칩을 지원하고 있음

약어 정리

ACL	Arm Compute Library
AI	Artificial Intelligence
API	Application Programming Interface
BLAS	Basic Linear Algebra Subprograms
CNN	Convolutional Neural Networks

CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DNN	Deep Neural Networks
GEMM	GEneral Matrix Multiplication
GPU	Graphics Processing Unit
HIP	Heterogeneous-computing Interface for Portability
HLO	High Level Operations
HPC	High Performance Computing
IR	Intermediate Representation
LLVM	Low Level Virtual Machine
LSTM	Long Short-Term Memory
MLIR	Multi-Level Intermediate Representation
NPU	Neural Processing Unit
ONNX	Open Neural Network eXchange
OpenCL	Open Computing Language
RNN	Recurrent Neural Network
ROCm	Radeon Open Compute stack
SDK	Software Development Kit
SIMD	Single Instruction Multiple Data
SSA	Static Single Assignment
STC	Systolic Tensor Core
SVE	Scalable Vector Extension
XLA	Accelerated Linear Algebra

참고문헌

[1] Y. Lecun et al., "Gradient-based learning applied to document recognition," Proceedings of the IEEE, vol. 86, no. 11, 1998, pp. 2278-2324.

[2] A. Krizhevsky, I. Sutskever, and G.E. Hinton, "Imagenet classification with deep convolutional neural networks," Adv. Neural Inf. Process. Syst. vol. 25, 2012, pp. 1097-1105.

[3] K. He et al., "Deep residual learning for image recognition," in Proc. Conf. Comput. Vis. Pattern Recognit. Las Vegas, NV, USA, June 2016.

[4] H. Pham et al., "Meta pseudo labels," arXiv preprint, CoRR, 2020, arXiv:2003.10580

[5] https://mlcommons.org/en/

[6] Arm, "Arm NN future roadmap," https://developer.arm.com/ip-products/processors/machine-learning/arm-nn

[7] https://github.com/ARM-software/ComputeLibrary

[8] https://github.com/NervanaSystems/ngraph

[9] Intel, "nGrapph developer guide," https://docs.openvinotoolkit.org/latest/openvino_docs_nGraph_DG_Introduction.html

[10] Android, https://developer.android.com/ndk/guides/neuralnetworks

[11] https://github.com/RadeonOpenCompute/ROCm

[12] https://github.com/NVIDIA/TensorRT

[13] https://github.com/Xilinx/Vitis-AI

[14] https://github.com/tensorflow/tensorflow

[15] https://github.com/pytorch/pytorch

[16] https://github.com/onnx/onnx

[17] https://github.com/microsoft/onnxruntime

[18] https://github.com/xianyi/OpenBLAS

[19] https://github.com/math-atlas/math-atlas

[20] https://github.com/oneapi-src/oneDNN

[21] https://developer.nvidia.com/cuda-toolkit

[22] https://developer.nvidia.com/CUDnn

[23] C. Nugteren, "CLBlast: A tuned OpenCL BLAS library," in Proc. Int. Workshop OpenCL, Oxford, UK, May 2018, 5:1-10.

[24] https://github.com/NVIDIA/cutlass

[25] https://www.tensorflow.org/xla

[26] C. Lattner et al., "MLIR: A compiler infrastructure for the end of Moore's law," arXiv preprint, CoRR, 2020, arXiv:2002.11054

[27] https://github.com/onnx/onnx-mlir

[28] T.D. Le et al., "Compiling ONNX neural network models using MLIR," arXiv preprint, CoRR, 2020, arXiv:2008.08272

[29] Y.C.P. Cho et al., "AB9: A neural processor for inference acceleration," ETRI J. vol. 42, no. 4, 2020, pp. 491-504.

[30] J. Han, M. Choi, and Y. Kwon, "40-TFLOPS artificial intelligence processor with function-safe programmable many-cores for ISO26262 ASIL-D," ETRI J. vol. 42, no. 4, 2020, pp. 468-479.

[31] H.M. Kim, C.G. Lyuh, and Y. Kwon, "Automated optimization for memory-efficient high-performance deep neural network accelerators," ETRI J. vol. 42, no. 4, 2020, pp. 505-517.

[32] 이미영 외, "인공지능프로세서 기술 동향," 전자통신동향분석, 제35권 제3호, 2020, pp. 66-75.

[33] 한진호, 권영수, "병렬 컴퓨팅 기반 인공지능 프로세서 기술 동향," IITP 주간기술동향, 제1964호, 2020, pp. 16-29.