ISSN 2383-630X(Print) / ISSN 2383-6296(Online)
Journal of KIISE, Vol. 46, No. 3, pp. 219-227, 2019. 3
https://doi.org/10.5626/JOK.2019.46.3.219

ARM 기반 IoT 장치에서 효율적인 딥 러닝 수행을 위한 BLAS 및 신경망 라이브러리의 성능 및 에너지 비교

(Performance and Energy Comparison of Different BLAS and Neural Network Libraries for Efficient Deep Learning Inference on ARM-based IoT Devices)

이 하 윤 † 신 동 군 ††
(Hayun Lee) (Dongkun Shin)

요 약 기존에 IoT 장치에서 딥 러닝을 수행하기 위해 주로 클라우드 컴퓨팅을 사용했다. 그러나 클라우드 컴퓨팅을 사용할 경우 연결을 보장할 수 없고, 통신을 위한 에너지 소모, 그리고 보안에 대한 취약성이 문제가 된다. 이와 같은 문제점을 해결하기 위해 최근 IoT 장치 내에서 딥 러닝을 수행하기 위한 시도가 진행되고 있다. 이 시도들은 주로 IoT 장치를 위한 연산량이 적은 딥 러닝 모델 또는 압축 기법 등을 제안하지만, 실제 IoT 장치에서 수행될 때의 영향에 대한 분석이 부족했다. IoT 장치마다 연산 장치의 구성과 지원되는 라이브러리가 다르기 때문에, 최적의 딥 러닝 수행을 위해 각 IoT 장치에서 다양한 수행 환경에 대한 분석이 필요하다. 본 논문에서는 다양한 하드웨어 구성을 가진 IoT 장치에서 수행 환경에 따른 성능 및 에너지를 측정하고 분석한다. 또한, 적절한 라이브러리를 사용하는 것만으로도 속도와 에너지 효율이 최대 9.43배, 26.78배까지 상승하는 것을 보여준다.

키워드: IoT, 딥 러닝, 딥 러닝 라이브러리, 딥 러닝 프로파일링, GPU 가속

Abstract Cloud computing is generally used to perform deep learning on IoT devices. However, its application is associated with limitations such as connection instability, energy consumption for communication, and security vulnerabilities. To solve such problems, recent attempts at performing deep learning within IoT devices have occurred. These attempts mainly suggest either lightweight deep learning models or compression techniques concerning IoT devices, but they lack analysis of the effect when it is performed in actual IoT devices. Since each IoT device has different configuration of processing units and supported libraries, it is necessary to analyze various execution environments in each IoT device in order to perform optimized deep learning. In this study, performance and energy of IoT devices with various hardware configurations were measured and analyzed according to the application of the deep learning model, library, and compression technique. It was established that utilizing the appropriate libraries improve both speed and energy efficiency up to 13.3 times and 48.5 times, respectively.

Keywords: IoT, deep learning, deep learning library, deep learning profiling, GPU-based acceleration

· 이 논문은 2019년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임(No.IITP-2017-0-00914, 지능형 IoT 장치용 소프트웨어 프레임워크)
· 이 논문은 2018 한국컴퓨터종합학술대회에서 'ARM 기반 IoT 장치에서 효율적인 딥 러닝 수행을 위한 BLAS 및 신경망 라이브러리의 성능 및 에너지 비교'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : 성균관대학교 전자전기컴퓨터공학과
lhy920806@gmail.com
†† 종신회원 : 성균관대학교 전자전기컴퓨터공학과 교수
(Sungkyunkwan Univ.)
dongkun@skku.edu
(Corresponding author임)

논문접수 : 2018년 8월 22일
(Received 22 August 2018)
논문수정 : 2018년 11월 26일
(Revised 26 November 2018)
심사완료 : 2018년 12월 4일
(Accepted 4 December 2018)

Copyright©2019 한국정보과학회 : 개인 목적이나 교육 목적인 경우, 이 저작물의 전체 또는 일부에 대한 복사본 혹은 디지털 사본의 제작을 허가합니다. 이 때, 사본은 상업적 수단으로 사용할 수 없으며 첫 페이지에 본 문구와 출처를 반드시 명시해야 합니다. 이 외의 목적으로 복제, 배포, 출판, 전송 등 모든 유형의 사용행위를 하는 경우에 대하여는 사전에 허가를 얻고 비용을 지불해야 합니다.

정보과학회논문지 제46권 제3호(2019. 3)

1. 서 론

딥 러닝은 이미지 분류, 자연어 처리, 자율주행차 등 다양한 분야에서 우수한 성능을 보인다. 최근 연구에서 고성능 GPU 또는 가속기에서 딥 러닝 수행을 가속하기 위해 많은 알고리즘 및 하드웨어 구조가 제안되었다. 그러나 기존 IoT 장치에서 딥 러닝을 가속하기 위한 연구는 상대적으로 적다.

이전에는 IoT 장치에서 딥 러닝 응용을 수행하기 위해 주로 클라우드 컴퓨팅을 사용했다. 그러나 클라우드 컴퓨팅을 사용하게 되면 세 가지 문제점이 존재한다. 첫번째, 불안정한 통신환경과 예측 불가능한 응답 시간으로 인해 서비스 제공에 차질이 발생할 수 있다. 두 번째, 통신을 위해 필요한 에너지가 IoT 장치만으로 수행할 때보다 클 수 있다[1]. 마지막으로, 클라우드로 데이터를 올려야 하므로 보안에 취약하다. 이와 같은 문제점들을 해결하기 위해 IoT 장치에서 딥 러닝을 수행할 필요가 있다.

IoT 장치는 제한된 전력과 낮은 계산 성능을 가지고 있다. 이전 연구들에서는 IoT 장치에서 딥 러닝을 수행하기 위하여 최적화된 네트워크 모델[2,3]과 네트워크 압축 기법 등이 제안되었다. 이들은 모두 딥 러닝 수행을 위한 모델의 크기와 곱셈 연산의 수를 줄이는 것에 초점을 맞추었다.

일반적으로 IoT 장치에서 딥 러닝 모델을 수행하기 위해 Caffe, TensorFlow와 같은 딥 러닝 프레임워크를 사용한다. 이러한 딥 러닝 프레임워크들은 CPU 또는 GPU에서 딥 러닝 수행을 가속하기 위해 각 장치에 최적화된 BLAS(Basic Linear Algebra Subprograms) 또는 신경망 라이브러리를 사용한다.

그러나 딥 러닝 수행을 위해 사용할 수 있는 BLAS 및 신경망 라이브러리는 IoT 장치에 따라 다르다. 또한 같은 구조의 네트워크 모델을 사용하더라도, 수행하는 라이브러리에 따라 수행 시간과 에너지가 달라진다. 그러므로 효율적인 딥 러닝 수행을 위해 각 IoT 장치에서 다양한 수행 환경에 대한 분석이 필요하다. 그러나 이전 연구에서는 이와 같은 분석이 제대로 이루어지지 않았다.

본 논문에서는 다양한 IoT 장치, 딥 러닝 모델, 라이브러리에 대해 수행 시간 및 에너지를 측정한다. 측정에 사용되는 장치들은 표 1과 같이 다양한 하드웨어 구성과 특성을 갖는다. 이후 장치-라이브러리 별 성능 분석을 통해 기존 대비 속도, 에너지 효율을 비교 및 분석한다.

2장에서는 본 논문과 관련된 연구들을 소개한다. 3장에서는 CNN 및 라이브러리와 관련된 배경 지식을 설명한다. 4장에서는 실험 환경 및 실험 결과를 설명하고, 5장에서는 결론을 제시한다.

표 1 다양한 하드웨어 구성 및 지원을 가진 IoT 장치들
Table 1 IoT devices with various hardware configurations and support

	Raspberry Pi 3 *(PI3)*	ODROID XU3 *(XU3)*	Jetson TX2 *(TX2)*
CPU	Cortex-A53 (quad core)	Cortex-A15 (quad core)	Cortex-A57 (quad core)
		Cortex-A7 (quad core)	Denver2 (dual core)
GPU		Mali T-628 MP6	256-core Pascal
NEON	✓	✓	✓
OpenCL		✓	
CUDA			✓

2. 관련 연구

CNN(Convolutional Neural Network)은 이미지 분류 및 센서 데이터 처리와 같은 작업에서 우수한 성능을 보여준다. 그러나 초기 CNN 모델은 메모리 요구량과 연산량이 많아 제한된 자원을 가진 IoT 장치에서 사용하기에 적합하지 않았다. 이를 해결하기 위해 Squeeze Net[2], MobileNet[3]과 같은 경량 CNN 모델이 제안되었다. 이들은 각각 fire 모듈과 depthwise separable convolution을 사용해 이전 CNN 모델의 정확도를 유지하면서도 연산량과 모델의 크기를 크게 줄였다.

이전부터 서로 다른 BLAS 라이브러리를 비교하는 연구는 존재했다. CPU와 GPU에서 CLBlast, clBLAS, cuBLAS를 사용하여 BLAS 라이브러리의 수행시간을 측정한 연구의 경우, 다양한 아키텍처에 따라 각각의 라이브러리의 성능을 보여준다[4]. 또한 CPU와 GPU에서 각각 MKL, cuBLAS를 사용하여 수행시간을 분석한 연구도 있다[5]. 그러나 이들은 세 가지 한계점이 존재한다. 첫번째, CLBlast와 clBLAS 실험을 제외하고는 ARM 기반 IoT 장치에서의 BLAS 비교를 수행하지 않았다. 두번째, 수행하는 동안 소모하는 에너지에 대한 고려가 되어있지 않다. 세번째, 실제 BLAS 라이브러리를 사용하여 딥 러닝 모델을 수행하는 환경에 대한 고려가 없었다.

다양한 장치에서 딥 러닝 모델의 수행을 비교하는 연구도 있었다[6]. 이 연구에서는 Snapdragon 800, Tegra K1, Edison에 대해 딥 러닝의 수행 시간과 에너지 소모량을 측정하였다. 그러나 이 논문은 두 가지 한계점이 있다. 첫번째, 다양한 라이브러리를 고려하지 않고 딥 러닝 수행 환경을 고정하였다. 두번째, Tegra K1을 제외한 나머지 장치들은 라이브러리 지원이 없어 직접 구현하여 실험하였다.

본 논문에서는 앞서 언급된 한계점들을 해결하기 위

해 ARM 기반 IoT 장치에서 사용될 수 있는 다양한 BLAS 및 신경망 라이브러리에 대하여 딥 러닝 모델 수행 단위로 수행 시간과 에너지 소모량을 비교하여 결과를 분석한다.

3. 배 경

3.1 CNN(Convolutional Neural Network)

CNN은 convolution 레이어를 사용하는 신경망을 의미한다. CNN은 사진, 음성과 같이 지역성(locality)이 있는 데이터를 처리함에 있어 좋은 성능을 보여주기 때문에 RNN(Recurrent Neural Network)와 함께 딥 러닝에서 많이 사용되고 있다.

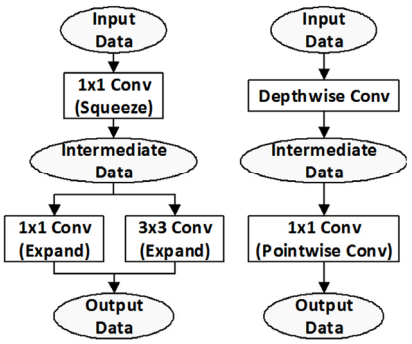
Convolution 레이어의 배치 크기가 1이라 가정할 때, 입력 데이터 $\boldsymbol{I} \in \mathbb{R}^{C \times H \times W}$ 와 해당 레이어의 커널 $\boldsymbol{K} \in \mathbb{R}^{N \times C \times K_h \times K_w}$ 를 통해 출력 데이터 $\boldsymbol{O} \in \mathbb{R}^{N \times H \times W}$ 를 만든다. 이때, C, H, W 는 각각 입력 데이터의 채널, 높이, 너비를 나타내고, N, K_h, K_w 는 각각 커널의 개수, 높이, 너비를 나타낸다. Convolution 레이어의 연산을 수식으로 나타내면 다음과 같다.

$$\mathbf{O}(n,h,w) = \sum_{c=0}^{C-1} \sum_{k_h=0}^{K_h-1} \sum_{k_w=0}^{K_w-1} \boldsymbol{K}(n,c,k_h,k_w) \cdot \boldsymbol{I}(c,h+k_h,w+k_w) \tag{1}$$

초기에 제안되었던 CNN 모델은 주로 이미지의 특징을 추출하고, 분류하기 위해 convolution 레이어와 fully-connected 레이어를 같이 사용하였다. 그러나, fully-connected 레이어를 사용할 경우 두가지 문제점이 존재한다. 첫번째, convolution 레이어와 비교하여 많은 수의 파라미터가 필요하다. 두번째, 입력 데이터의 크기가 고정된다. 이러한 문제를 해결하기 위해 최근 fully-connected 레이어를 사용하지 않은 CNN 모델들이 많이 나타나고 있다[2,3]. 이 모델들은 전체를 convolution 레이어로 구성하고, 마지막 convolution 레이어 이후에 global average pooling을 통해 fully-connected 레이어를 대체한다.

일반적으로 CNN 모델은 상당히 많은 수의 파라미터와 곱셈 연산을 필요로 하기 때문에 IoT 장치와 같이 자원이 제한된 모바일 환경에 적용시키기 어려웠다. 그러나 최근 연구에서 SqueezeNet, MobileNet과 같이 모바일 환경을 위한 모델들이 제안되어, 적은 연산으로 기존의 모델들과 근접한 성능을 내는 것이 가능하게 되었다. SqueezeNet과 MobileNet은 모두 fully-connected 레이어가 존재하지 않으며, 그림 1과 같은 모듈을 기반으로 구성된다.

그림 1(a)는 SqueezeNet에서 사용되는 fire 모듈이다. Fire 모듈은 squeeze 레이어와 expand 레이어로 구성된다. Squeeze 레이어에서는 1x1 convolution을 통해 입력 데이터보다 적은 채널 수를 가지는 중간 데이터를 만듦으로써 이후 convolution 수행에 필요한 연산의 수를 줄인다. Expand 레이어에서는 중간 데이터를 1x1, 3x3 convolution을 통해 결과 데이터를 증가시킨다.



(a) Fire Module (b) Depthwise Separable Convolution

그림 1 다양한 CNN 모듈

Fig. 1 Various CNN modules

그림 1(b)는 MobileNet에서 사용되는 DSC (Depthwise Separable Convolution)이다. DSC는 depthwise convolution과 pointwise convolution으로 구성된다. Depthwise convolution은 각 채널마다 convolution을 수행하여 연산량을 줄인다. Pointwise convolution은 1x1 convolution과 같으며 depthwise convolution의 결과들을 합치기 위해 수행한다. DSC를 사용할 경우 기존과 비교하여 연산량은 다음과 같은 감소를 얻게 된다.

$$\frac{K_h \times K_w \times C \times H \times W + C \times N \times H \times W}{K_h \times K_w \times C \times N \times H \times W} = \frac{1}{\mathrm{N}} + \frac{1}{K_h K_w} \approx \frac{1}{K_h K_w} \tag{2}$$

위의 식에서 N은 depthwise convolution의 커널의 크기인 K_h, K_w 보다 상대적으로 크기 때문에 무시 가능하다. 즉, DSC는 depthwise convolution의 커널을 3x3으로 사용하였을 때, 기존의 convolution과 비교하여 연산량이 약 9배 감소하게 된다.

과거에는 convolution 레이어 연산자체를 최적화하는 시도가 거의 없었기 때문에, convolution 레이어를 구현하기 위해 주로 최적화가 잘 되어있는 BLAS 라이브러리의 GEMM(GEneral Matrix Multiplication) 연산을 사용하였다. 그림 2와 같이 입력 데이터에 im2col 연산을 수행한 후 GEMM을 수행하면 convolution과 같은 결과를 얻을 수 있다. 최근에 딥 러닝을 위한 신경망 라이브러리들이 개발되면서 BLAS 라이브러리를 사용하여 수행한 결과와 비교하여 더 빠른 수행 시간을 제공하기도 한다.

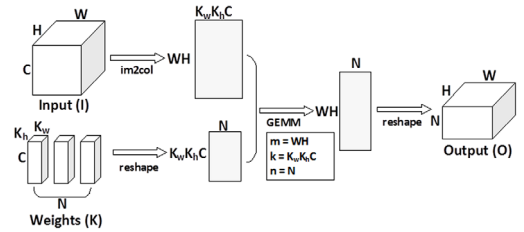


그림 2 GEMM을 사용한 convolution 연산
Fig. 2 Convolution operation using GEMM

3.2 BLAS / 신경망 라이브러리

그림 3은 본 논문에서 딥 러닝 모델 수행을 실시한 소프트웨어 스택을 보여준다. 굵은 상자는 BLAS 및 신경망 라이브러리를 나타내며, 이름 아래의 괄호는 본 논문에서 사용된 버전을 의미한다. 라이브러리마다 지원하는 연산 장치가 다르기 때문에, 각 장치에 맞는 라이브러리를 사용해야 한다. 딥 러닝 모델 수행 시 ArmCL[7]과 TVM[8]을 사용하는 경우를 제외하고는 딥 러닝 프레임워크로써 Caffe[9]를 사용했다. 또한 그림 3에서의 Caffe와 Caffe^{opencl}은 각각 github에서 master 브랜치(commit: 8645207)와 opencl 브랜치(commit: 74312cf)를 의미한다.

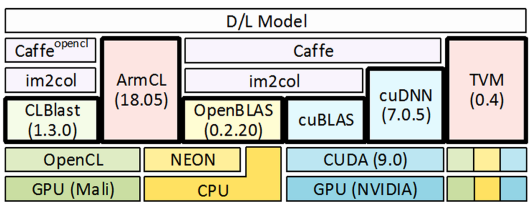


그림 3 딥 러닝 모델 수행을 위한 소프트웨어 스택
Fig. 3 Software stack for deep learning model execution

3.2.1 BLAS 라이브러리

BLAS는 벡터와 매트릭스 연산을 수행하기 위해 최적화된 함수를 제공하는 라이브러리이다. 레벨 1부터 3까지 각각 벡터-벡터, 벡터-매트릭스, 매트릭스-매트릭스 연산을 수행하는 함수가 제공된다. BLAS는 오랜 시간 다양한 하드웨어에 맞게 최적화되어왔기 때문에 딥 러닝 수행을 위해서도 많이 사용된다. 예를 들어, Caffe는 im2col 함수와 함께 BLAS 라이브러리를 사용하여 convolution 레이어를 구현한다. 대표적인 BLAS 라이브러리에는 OpenBLAS[10], CLBlast[4], cuBLAS가 있다.

OpenBLAS는 다양한 CPU 특징들에 맞게 최적화된 BLAS 라이브러리이며, 멀티 쓰레드를 통한 가속을 지원한다. 본 논문에서 타겟으로 하고 있는 ARMv7, ARMv8 CPU에 대해서도 최적화가 되어있으며, ARMv8부터 NEON을 지원(ARMv7의 경우 성능상의 문제로 지원 안함)한다.

CLBlast는 OpenCL을 지원하는 CPU 및GPU에 최적화된 BLAS 라이브러리이다. CLBlast는 다양한 장치에 대해 최적화가 가능하도록 튜닝할 수 있는 파라미터를 제공한다. XU3의 경우 CLBlast를 사용하여 GPU 가속이 가능하지만, TX2에서는 사용할 수 없다. 그 이유는 TX2의 GPU가 OpenCL을 지원하지 않기 때문이다.

cuBLAS는 CUDA를 지원하는 GPU에 최적화된 BLAS 라이브러리이다. cuBLAS를 통해 TX2에 내장된 GPU를 가속할 수 있다.

3.3.2 신경망 라이브러리

신경망 라이브러리는 BLAS 라이브러리와 다르게 딥 러닝 워크로드에 최적화된 함수를 제공한다. 예를 들어, 신경망 라이브러리는 convolution 레이어를 수행하기 위해 GEMM 뿐만이 아니라 direct convolution, FFT, Winograd 등 convolution을 빠르게 수행하기 위한 다양한 알고리즘들을 제공한다. 대표적인 신경망 라이브러리에는 ArmCL(ARM Compute Library)[7], cuDNN이 있다.

ArmCL은 ARM CPU와 GPU에 최적화된 신경망 라이브러리이다. ARM CPU와 GPU에 대하여 각각 NEON, OpenCL로 가속을 수행한다. 또한, OpenCL로 가속할 경우 CLTuner를 통해 최적화된 OpenCL 커널 및 워크그룹 크기를 찾아준다. ArmCL은 Caffe와 같은 딥 러닝 프레임워크 없이 딥 러닝 수행이 가능하며, ArmNN을 사용할 경우 Caffe 또는 TensorFlow 모델을 ArmCL을 사용하여 딥 러닝을 수행할 수 있다.

cuDNN은 CUDA를 지원하는 GPU에 최적화된 신경망 라이브러리이다. ArmCL과 유사하게 레이어 별로 API를 제공한다.

3.3.3 TVM

TVM[8]은 CPU, GPU, FPGA 등 다양한 장치에 대한 최적화된 딥 러닝 수행 코드를 생성하는 딥 러닝 컴파일러 스택이다. TVM 스택은 계산 그래프 최적화(computational graph optimization)와 텐서 연산 스케줄링(tensor operation scheduling)을 통해 코드를 생성한다.

TVM 버전 0.4에서 AUTOTVM 모듈이 추가되었으며, 해당 모듈을 통해 스케줄링 공간 상의 여러 경우의 수를 시도하여 최적의 수행 코드를 찾는다. 해당 버전까지는 ARM CPU에 대해서만 지원이 되고, 버전 0.5부터 OpenCL과 CUDA를 지원할 예정이다.

4. 실험

4.1 실험 환경

실험을 위해 표 1에서 언급된 세 가지 장치와 그림 3에 있는 여섯 가지 라이브러리를 사용하였다. 라이브러

리마다 지원하는 장치가 다르기 때문에, 각 장치에 맞는 라이브러리만 사용하여 실험을 진행하였다.

전력 측정을 위해 PI3에서는 Monsoon Power Monitor를 사용하였고, XU3와 TX2에서는 내장된 power monitor를 사용하였다. OpenBLAS의 스레드 개수는 모두 4개로 통일하였고, TX2의 Denver2 CPU는 활성화하지 않은 상태에서 실험을 진행하였다.

CNN 실험을 위해 SqueezeNet(Sq) v1.1과 Mobile-Net (Mo) v1을 사용하였다.

4.2 GEMM 성능 비교

그림 4는 매트릭스 크기에 따라 각 장치-라이브러리 별 SGEMM(Single precision floating GEMM) 연산 속도를 보여준다. $m \times k$, $k \times n$ 크기를 갖는 매트릭스 두개에 대하여 GEMM 연산을 한다고 가정했을 때, FLOPS 산출 방식은 다음과 같다.

$$\frac{m \times n \times (2k-1)}{\text{execution_time}} \tag{3}$$

그림 4에서, 모든 라이브러리는 매트릭스 크기가 커짐에 따라 연산 속도도 같이 증가한다. OpenBLAS는 매트릭스 크기가 작을 때에는 빠르지만, 매트릭스 크기가 커짐에 따라 속도 향상이 크지 않음을 볼 수 있다. 또한, CLBlast 실험 결과를 제외하면 GPU로 수행하는 것이 CPU로 수행하는 것보다 빠른 성능을 보여준다. 특히, TX2에서 cuBLAS를 사용할 때 다른 라이브러리와 비교하여 독보적으로 빠른 성능을 보인다.

4.3 CNN 수행 성능 비교

그림 5와 6은 각각 장치-라이브러리별 CNN 수행에 대한 수행 시간 및 에너지 효율과 전력 소모를 보여준다.

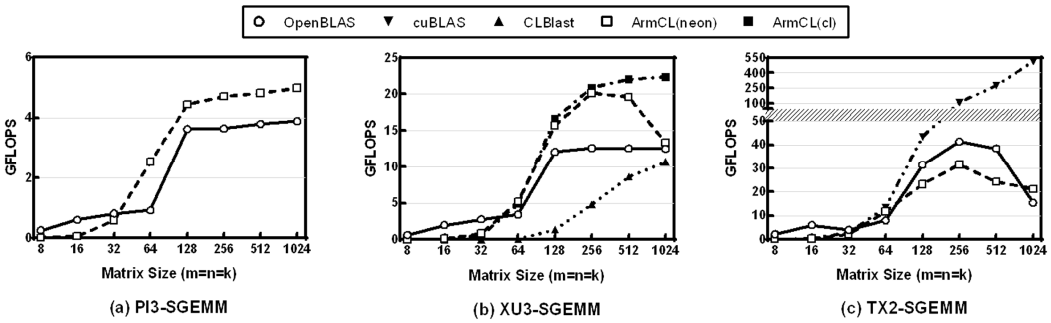


그림 4 장치-라이브러리 별 SGEMM 수행 성능 비교

Fig. 4 Comparison of devices and libraries in SGEMM execution performance

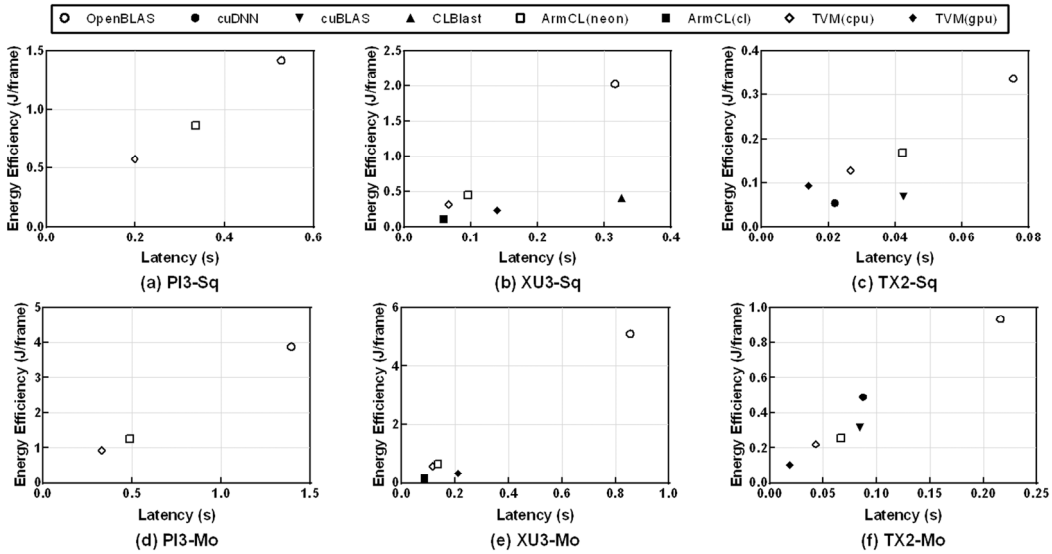
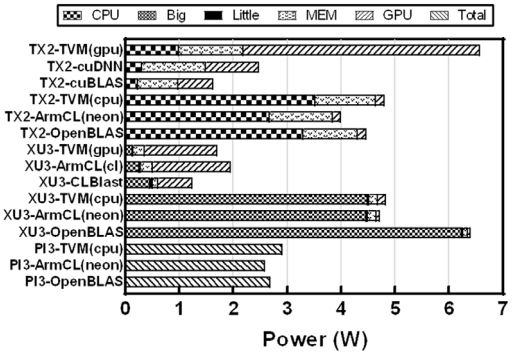
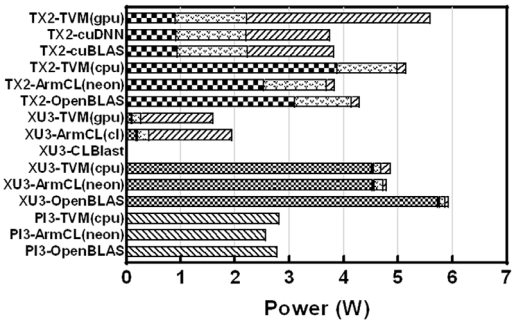


그림 5 장치-라이브러리 별 CNN 모델 수행 시간 및 에너지 효율 비교

Fig. 5 Comparison of devices and libraries in CNN model execution time and energy efficiency



(a) Sq



(b) Mo

그림 6 장치-라이브러리별 CNN 모델 수행에 따른 전력 소모

Fig. 6 Comparison of power consumption according to CNN model execution for different device-library pairs

그림 5에서 흰색 심볼은 CPU, 검은색 심볼은 GPU에서 수행되었음을 의미한다. ArmCL(cl)과 TVM(cpu)는 모두 튜닝을 수행한 후 실행한 결과이고, 그 외의 라이브러리는 튜닝이 지원되지 않는다. 그림 6의 Big/Little은 각각 XU3의 A15/A7 CPU의 전력 소모를 나타낸다. XU3-Mo의 CLBlast 실험은 메모리 부족으로 실행 불가능하여 제외하였다.

그림 5에서 PI3는 Sq와 Mo의 라이브러리 수행시간 순서가 같고, XU3도 마찬가지이다. 그러나 TX2에서는 Sq와 Mo의 라이브러리 수행시간 순서가 다르다. TX2에서 Sq의 라이브러리 수행시간 순서는 cuDNN, ArmCL(neon), cuBLAS이지만, Mo일 때 수행시간 순서는 ArmCL(neon), cuBLAS 그리고 cuDNN으로 ArmCL(neon)이 가장 빠른 속도를 보이고, 에너지 효율도 ArmCL(neon)이 cuBLAS와 cuDNN보다 좋다. cuDNN은 Winograd[11]와 같은 빠른 convolution 알고리즘과 그 외에도 최적화된 커널들을 수행할 수 있기 때문에 보통 cuBLAS로 수행하는 것보다 빠르다. 그러나 cuDNN

```
// G: number of groups
void ConvolutionLayer::Forward_gpu() {
    for (int g = 0; g < G; g++) {
        im2col_gpu(g);
        cublasSgemm(g);
    }
}

void CuDNNConvolutionLayer::Forward_gpu()
{
    for (int g = 0; g < G; g++) {
        cudnnConvolutionForward(g);
    }
}
```

그림 7 Caffe 프레임워크의 Convolution 구현

Fig. 7 Convolution implementation in Caffe framework

이 Mo에서 이들 중 가장 느린 이유는 Mo에서 사용하는 depthwise convolution을 Caffe에서 비효율적으로 수행하기 때문이다.

그림 7은 Caffe의 convolution 구현을 간단하게 나타낸 것이다. Caffe에서는 depthwise convolution을 수행하기 위해 그룹의 개수 G가 입력의 채널 수와 같은 convolution을 수행한다. Caffe의 구현 상 cuBLAS와 cuDNN 버전 모두 그룹의 수만큼 함수를 반복 호출하게 된다. 즉, 하나의 convolution을 여러 개의 CUDA 커널로 나누어 수행하면서 연산을 수행하는 데이터의 크기가 작아지게 되므로 그림 4와 같이 전체적으로 연산 수행 속도가 감소한다. 그러나 ArmCL(neon)의 경우 depthwise convolution를 하나의 커널로 수행할 수 있도록 구현되어 있기 때문에 둘과 비교하여 좋은 성능을 보여준다. 또한 실험에 사용된 모든 장치에서 OpenBLAS 대신 ArmCL(neon) 또는 TVM(cpu)를 사용했을 때, Sq보다 Mo에서 수행시간 향상이 더 큰 이유도 앞에서 말한 Caffe의 구현 때문이다. 추가적으로 cuDNN은 입력의 채널이 1인 convolution을 수행할 경우 GEMV(Matrix-Vector) 연산을 수행하므로 cuBLAS와 유사한 성능을 내게 된다.

그림 5에서 GPU로 수행했을 경우 TX2-Mo 실험을 제외했을 때 대체적으로 수행시간과 상관없이 CPU 수행과 비교하여 에너지 효율이 좋다. 이는 그림 5와 6을 통해 두가지 분석이 가능하다. 첫번째, XU3는 GPU의 전력 소모가 CPU 전력 소모와 비교하여 최소 19.4%에서 최대 41.5% 밖에 되지 않기 때문이다. 단, CLBlast는 전력 소모 자체는 적으나, 수행 시간이 느리기 때문에 에너지 효율이 좋지 않다. 두번째, TX2의 경우 CPU와 동일한 수행 시간이 걸리더라도 전력 소모가 낮거나, TVM(gpu)와 같이 전력 소모가 크더라도 수행 속도가 매우 빠르기 때문이다. 단, Mo에서 cuDNN과 cuBLAS는 CPU보다 수행 시간도 느리고, 전력 소모도 비슷하기 때문에 에너지 효율이 낮다.

마지막으로, XU3에서 TVM(gpu)가 TVM(cpu)보다 수행 시간이 작은데 이는 현재 TVM이 ARM CPU에 대해서만 auto-tuner를 지원하기 때문이다. TVM의 auto-tuner는 데이터 크기에 최적화된 파라미터 설정을 찾아 최적의 커널 코드를 생성하므로 높은 성능 향상을 보일 수 있다. 이에 반해 ArmCL(cl)에서도 튜닝은 지원하지만 이미 구현되어 있는 커널 코드 중에서 최적의 워크그룹 크기를 찾는 것이기 때문에 TVM과 차이가 있다.

4.4 Half-precision float-point 연산

기존의 single-precision float-point 연산 대신 half-precision float-point를 사용할 경우CNN의 수행 시간을 더 단축시킬 수 있다. 그림 8은 XU3에서 딥 러닝을 수행한 결과와 ArmCL(cl)-fp16의 수행 결과를 비교한다.

XU3에 내장되어 있는 Mali T-628은 shader core 안에 두 개의 arithmetic 파이프라인을 가지고 있다. Arithmetic 파이프라인은 128-bit quad-word 레지스터를 가지고 있는 SIMD (Single Instruction Multiple Data) 벡터 프로세싱 엔진이다. SIMD를 사용할 경우 FP32와 FP16은 각각 4개, 8개의 연산을 동시에 처리할 수 있다. 그러므로 FP16을 사용할 경우 FP32 성능과 비교하여 최대 2배 증가할 수 있다. 그러나 그림 8과 같이 ArmCL(cl)-fp16은 ArmCL(cl)과 비교하여 수행 시간이 약 1.5~1.57배 증가했다. 2배가 되지 않았던 이유에 대해서는 Mali GPU 프로파일링을 통해 분석하도록 한다.

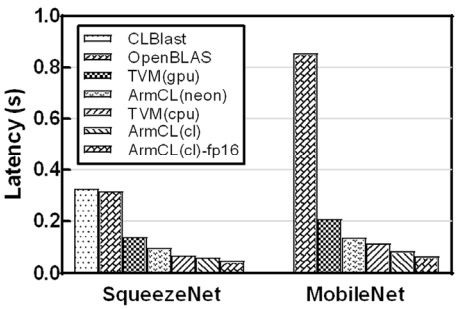


그림 8 XU3-FP16 수행시간 비교

Fig. 8 Comparison of XU3-FP16 execution time

4.5 Mali GPU 프로파일링

이번 실험에서는 ARM Streamline 도구를 사용하여 Mali GPU 프로파일링을 실시하였다. 그림 9는 XU3에서 Sq와 Mo를 수행하였을 때 이미지 한 장을 처리하기 위해 발생된 GPU 명령어 개수를 보여준다. 그래프의 A와 LS는 각각 Arithmetic과 Load/Store를 의미한다. 그림 8과 9를 보면 A 명령어와 LS명령어가 적을수록 수행시간이 빠르다는 것을 알 수 있다.

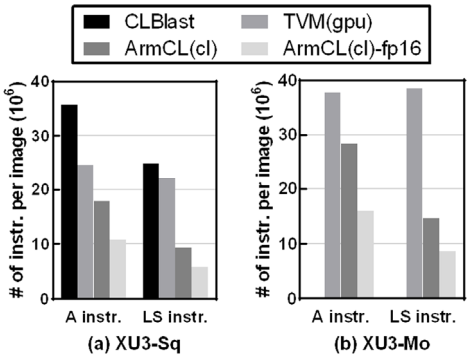


그림 9 이미지 한 장 처리당 GPU 명령어 개수

Fig. 9 Number of GPU instructions per one image processing

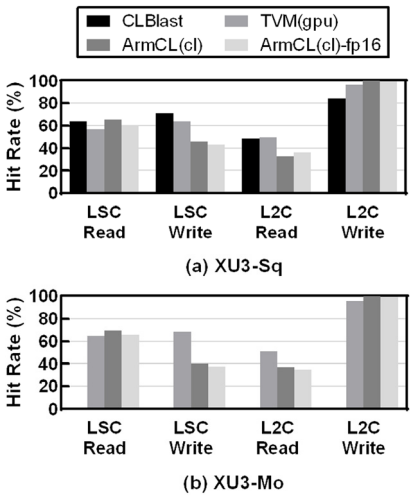


그림 10 XU3 캐시 hit rate

Fig. 10 XU3 cache hit rate

그림 10과 11은 각각 XU3에서 Sq와 Mo 수행 시 캐시 hit rate와 외부 버스(메인 메모리)로부터 읽고 쓰는 양을 보여준다. LSC는 LS 캐시를 의미하며 데이터를 읽고 쓰는 역할을 하는 LS 파이프라인의 L1 캐시이다. L2C는 L2 캐시를 의미하며 LS 파이프라인을 포함하고 있는 shader core의 캐시이다.

XU3는 ArmCL(cl)을 사용할 때 LSC 읽기와 L2C 쓰기의 hit rate가 높고, LSC 쓰기와 L2C 읽기의 hit rate가 낮음을 알 수 있다. 이를 통해 ArmCL(cl)이 TVM, CLBlast와 비교하여 L2 캐시 hit rate는 낮지만 메모리 접근을 최적화하여 구현했기 때문에 L2 캐시 접근 자체가 낮으므로 그림 11처럼 메인 메모리로부터 읽고 쓰는 양이 적음을 알 수 있다.

FP16 (ArmCL(cl)-fp16)이 FP32 (ArmCL(cl))와 비교 하여 시간이 2배로 줄지 않은 이유는 다음과 같다. 첫

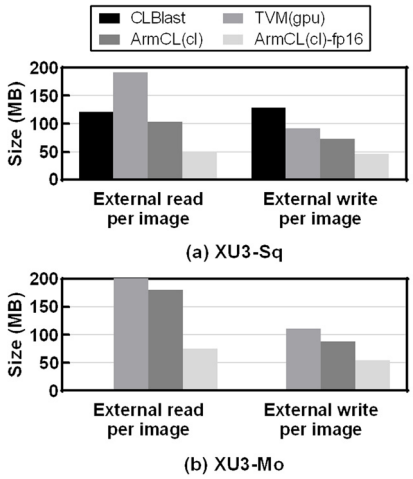


그림 11 XU3 외부 버스 읽기/쓰기
Fig. 11 XU3 external bus read/write

번째, 그림 9에서 알 수 있듯이 FP16의 A 명령어가 FP32보다 2배가 아닌 1.65~1.77배 적은 것을 확인할 수 있다. 왜냐하면 2배의 데이터를 동시에 연산한다고 해도 데이터의 끝부분에서 SIMD 공간을 충분히 활용하지 못하는 경우가 존재하기 때문이다. 두번째, FP16의 LS 캐시의 hit rate가 FP32의 것보다 낮다는 것은 L2캐시 접근이 2배로 줄어들지 않는 것을 의미하기 때문이다.

4.6 성능 향상 비교

표 2는 그림 5와 7의 결과를 바탕으로 수행 시간과 에너지 소모량 측면에서의 향상 결과를 보여준다. 베이스라인은 모두 Caffe 프레임워크에서 OpenBLAS 또는 cuDNN을 사용한 것으로 가정하였다. 이 때, 앞선 실험의 결과를 바탕으로 각 장치마다 속도 향상이 가장 높은 라이브러리를 선택하였다. TX2에서 Sq를 수행할 때를 제외하고는 속도와 에너지 효율이 모두 상승하였다.

표 2 최적화된 라이브러리를 통한 성능 향상
Table 2 Improved performance with optimized libraries

Device	RPI3		XU3		TX2	
Model	Sq	Mo	Sq	Mo	Sq	Mo
Baseline	Caffe + OpenBLAS / cuDNN					
Optimized	TVM(cpu)		ArmCL(cl)-fp16		TVM(gpu)	
Speed up	2.7x	4.3x	7.0x	13.3x	1.6x	4.7x
Energy	2.5x	4.2x	26.2x	48.5x	0.6x	3.1x

5. 결 론

본 논문은 딥 러닝을 수행하기 위한 다양한 라이브러리를 서로 다른 하드웨어 조건을 갖춘 IoT 장치에서 속도와 에너지를 비교 및 분석한다. 또한, 실험 결과를 통해 자신의 하드웨어에 맞는 최적의 라이브러리를 사용하는 것만으로도 속도와 에너지 효율이 최대 13.3배, 48.5배까지 상승하는 것을 보여준다.

References

[1] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *Proc. of the 23nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 615-629, 2017.

[2] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size," *CoRR*, abs/1602.07360, 2016.

[3] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, abs/1704.04861, 2017.

[4] C. Nugteren, "CLBLast: A tuned openCL BLAS library," *Proc. of the International Workshop on OpenCL*, Article No. 5, 2018.

[5] F. Li, Y. Ye, Z. Tian, and X. Zhang, "CPU versus GPU: which can perform matrix computation faster-performance comparison for basic linear algebra subprograms," *Neural Computing and Applications*, pp. 1-13, 2018.

[6] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, F. Kawsar, "An early resource characterization of deep learning on wearables, smartphones and internet-of-things devices," *Proc. of the 2015 International Workshop on Internet of Things towards Applications*, pp. 7-12, 2015.

[7] ARM. ARM Compute Library [Online]. Available https://github.com/ARM-software/ComputeLibrary (downloaded 2018, Dec. 11)

[8] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, A. Krishnamurthy, "TVM: End-to-End Optimization Stack for Deep Learning," *CoRR*, abs/1802.04799, 2018.

[9] The Berkeley Vision and Learning Center. Caffe [Online]. Available https://github.com/BVLC/caffe (downloaded 2018, Dec. 11)

[10] Zhang Xianyi. OpenBLAS [Online]. Available https://github.com/xianyi/OpenBLAS (downloaded 2018, Dec. 11)

[11] A. Lavin, S. Gray, "Fast algorithms for convolutional neural networks," *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4013-4021, 2016.

이 하 윤

2017년 성균관대학교 컴퓨터공학과(학사) 2017년~현재 성균관대학교 전자전기컴퓨터공학과 석박통합과정. 관심분야는 임베디드 시스템, IoT 프레임워크, 엣지 컴퓨팅

신 동 군

1994년 서울대학교 계산통계학과(학사) 2000년 서울대학교 전산과학과(석사). 2004년 서울대학교 컴퓨터공학부(박사). 2004년~2007년 삼성전자 소프트웨어 책임연구원 2007년~현재 성균관대학교 소프트웨어대학 부교수. 관심분야는 임베디드 시스템, 실시간 시스템, 저전력 시스템