Journal of The Institute of Electronics and Information Engineers Vol.57, NO.1, January 2020 https://doi.org/10.5573/ieie.2020.57.1.77
ISSN 2287-5026(Print) / ISSN 2288-159X(Online)

논문 2020-57-1-11

시스템 자원 관리를 통한 객체 인식 성능 저하 방지

(Avoiding Performance Degradation of Object Detection via System Resource Management)

김 명 선*

(Myungsun Kim©)

요 약

최근 딥러닝 기술의 빠른 발전 속도로 인하여 자율주행 자동차의 객체 인식 기능에서 인공 신경망의 사용이 보편화되고 있다. 더불어 높은 인식 정확도를 나타내기 위해 인공 신경망 연산의 복잡도와 데이터 요구량은 급속하게 늘고 있다. 인공 신경망의 연산을 클라우드 서버에서 수행 시 네트워크 지연으로 인한 응답특성이 떨어지거나 보안 문제의 여지가 있어서 자율주행 자동차 내부의 임베디드 시스템에서 수행한다. 최근 고 성능 멀티코어 기반 SoC(System on Chip) 기술 발전으로 신경망을 임베디드 환경에서 연산할 수 있게 되었다. 하지만 이러한 환경에서는 CPU나 메모리와 같은 시스템 자원을 다른 응용과 공유하기 때문에 객체 인식과 같은 안전에 민감한 기능이 수행될 때 성능 저하가 발생할 수 있다. 본 논문에서는 객체 인식 응용이 시작된 후 성능 간섭이 나타나면 객체 인식 응용에 관련된 모든 태스크들을 찾아내어 더 많은 시스템 자원 사용율을 부여하고 종료되면 원래 상태로 복귀되는 운영체제 수준의 솔루션을 제시한다. 제안된 기법을 Jetson AGX Xavier에 탑재 후 실험한 결과 객체 인식 성능이 SPEC CPU2017 벤치마크 프로그램들에게 성능 간섭을 받는 환경에서 적용 전 대비 13.5%~33% 향상되었다.

Abstract

Recently, due to the rapid development of deep learning technology, the use of deep neural networks in the object detection of autonomous vehicles is becoming prevalent. Besides, the complexity and data requirements of deep neural network operations are rapidly increasing to show high recognition accuracy. When deep neural network computation is performed in cloud server, response characteristics due to network delay is inferior or there is a possibility of security problem. Recent advances in high-performance, multicore-based System on Chip (SoC) technologies enable neural networks to be computed in embedded environments. However, in this environment, system resources such as CPU and memory are shared with other applications, which can cause performance degradation when safety-sensitive functions such as object detection are performed. In this paper, we propose an operating system level solution that finds all tasks related to object detection, gives more system resource utilization if the performance interference occurs after the object detection is started, and returns to the original state when finished. The proposed solution is running on top of Jetson AGX Xavier, and the experimental results show that the performance of object detection is improved by 13.5%~33% compared to before, under the performance interference of SPEC CPU2017 benchmark programs.

Keywords : Deep neural network, Object detection, System resource management, Mixed-criticality system

Ⅰ. 서 론

Mixed-Criticality 시스템에서는 각 응용에 부여된 중요도에 따라서 여러 수준의 응용 계층이 존재한다[1]. 이러한 시스템의 대표적인 예로 자율 주행 자동차를 들 수 있다. 자율주행 자동차에 있어서 객체 인식(Object Detection) 응용은 매 영상 프레임마다 객체를 실시간으로 판별하고 주행과 관련된 의사 결정을 수행할 수 있게 한다[2]. 이는 안전과 직결된 가장 중요한 기능으로서 가장 최상위 계층의 중요도를 가진다. 따라서 객체 인식 응용은 시스템 내 다른 어떤 계층의 응용들보다도 빠른 응답특성과 가장 짧은 수행시간 지연 특성을 나타내어야 한다.

*정회원, 한성대학교 IT융합공학부(Department of IT Convergence Engineering, Hansung University)
©Corresponding Author(E-mail : kmsjames@hansung.ac.kr)
※ "본 연구는 한성대학교 교내학술연구비 지원과제임."
Received ; August 29, 2019 Revised ; September 29, 2019
Accepted ; November 24, 2019

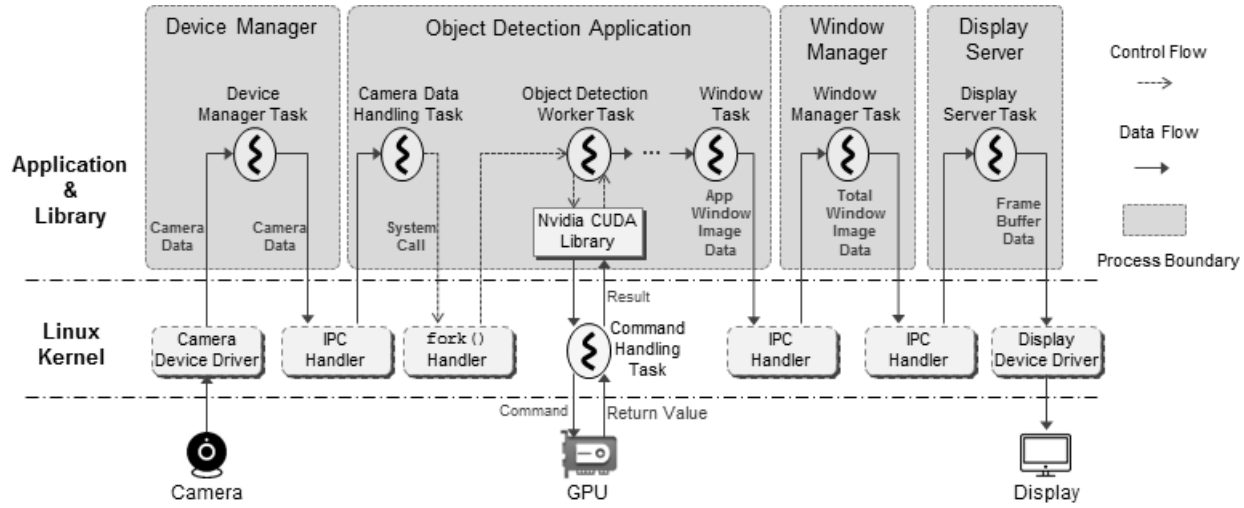

그림 1. 객체 인식 전체 과정
Fig. 1. Whole process of object detection.

자동차 내부에 사용되는 CPU 역시 기존의 분리된 많은 수의 저 사양 마이크로 컨트롤러 구조에서 통합된 고 성능 멀티코어 프로세서 형태로 발전되어 왔다[3]. 이러한 통합된 멀티코어 프로세서 환경에서는 객체 인식과 같은 최상위 중요도를 가진 응용과 하위 중요도를 갖는 응용들이 CPU나 메모리와 같은 시스템 자원을 공유하면서 수행된다. 이로 인하여 객체 인식 응용이 시스템 자원을 충분하게 사용하지 못할 경우 성능 저하가 발생할 수 있다[3].

최근 딥러닝 기술이 빠른 속도로 발전하고 있어서 DNN(Deep Neural Network)으로 대표되는 인공 신경망 기술이 객체 인식의 정확도를 매우 높였다[4~7]. 따라서 자율주행 자동차에서도 이러한 DNN 기반 객체 인식을 채택하고 있다[2, 4]. 하지만 DNN이 수행하는 연산의 성격은 그 계산량이 크고 실시간 처리를 위한 고속 연산이므로 시스템 자원을 공유하는 환경에서는 다른 응용들로부터 성능 간섭을 받을 수밖에 없다.

시스템 자원은 크게 CPU, 메모리, IO, 네트워크 등으로 나눌 수 있다. 본 연구에서 다루는 시스템은 DNN의 추론(Inference)을 수행하는 자율주행 자동차와 같은 임베디드 시스템이다. 따라서 클라우드 서버에서 추론을 수행하는 DNN 연산이 아니라서 네트워크 의존성이 없고 DNN 연산 특성 상 파일 IO가 많은 부하를 차지하지 않는다. 결과적으로 연구 대상으로 하는 시스템의 자원은 CPU와 메모리를 뜻한다.

본 논문에서는 객체 인식 응용이 시스템 자원을 다른 응용들과 공유하면서 운용될 때 성능 저하를 최소화하는 운영체제 수준의 솔루션을 제안한다.

Ⅱ. 객체 인식 과정 및 지연 요소

본 절에서는 객체 인식을 수행하는 전체 과정을 설명하고 성능 저하의 원인이 되는 부분을 설명한다.

1. 객체 인식 수행 과정

그림 1은 NVIDIA(사) GPU와 CUDA[8] 라이브러리를 사용한 객체 인식 수행 과정을 보여준다. 그림에서처럼 소프트웨어 부분은 응용과 라이브러리 계층 그리고 리눅스 커널 계층 이렇게 두 부분으로 나눌 수 있고 이는 모두 멀티코어로 구성된 CPU에서 수행된다.

최초 카메라로부터 영상이 입력되면 이는 디바이스 매니저 태스크로 전송되고 이는 리눅스 커널의 IPC(Inter Process Communication)를 통하여 객체 인식응용의 메인 태스크로 전달된다. 메인 태스크는 fork() 시스템 호출을 사용하여 여러 개의 워커 태스크를 생성하고 이들은 CUDA 라이브러리를 사용하여 DNN 연산에 필요한 명령과 데이터를 GPU에게 전달한다. GPU는 여러 개의 행렬 및 벡터의 곱셈으로 이루어진 DNN 연산을 수행한 후 그 결과 값을 명령어 처리 태스크(Command Handling Task)를 통하여 워커 태스크들에게 전달한다. 이후 윈도우 매니저는 화면에 표시할 전체 영상 데이터를 구성하고 이 데이터를 디스플레이 서버에 전달하여 프레임 데이터를 생성한 후 화면에 표시한다.

위에서 살펴본 것처럼 전체 객체 인식 과정은 GPU를

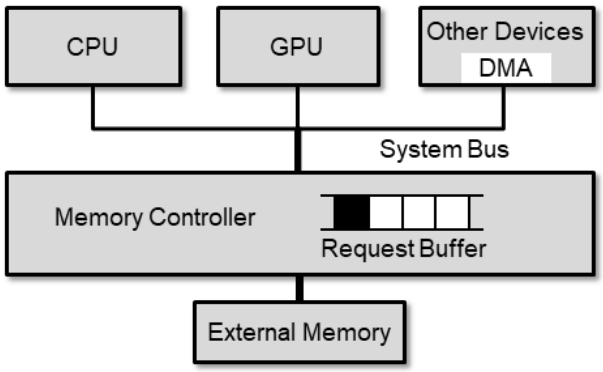

그림 2. 메모리 서브시스템
Fig. 2. Memory subsystem.

사용하는 DNN 연산뿐만 아니라 디바이스 매니저, 윈도우 매니저, 디스플레이 서버 등의 시스템 서버로 구성됨을 알 수 있다. 따라서 객체 인식 응용의 성능은 사용되는 시스템 서버와 객체 인식 응용의 메인 태스크가 생성하는 워커 태스크들을 포함한 전체의 성능이라고 할 수 있다.

2. 메모리 간섭에 의한 지연

그림 2는 일반적인 임베디드 시스템의 메모리 서브시스템을 나타낸다. CPU, GPU 혹은 디바이스의 DMA(Direct Memory Access) 등은 시스템 버스를 통하여 메모리 컨트롤러에게 메모리 요구 데이터를 전송하고 이들은 버퍼에 순차적으로 쌓인다.

객체 인식에 사용되는 DNN 연산은 많은 파라미터를 사용하는 메모리 집중적인 특징을 갖는다[9, 10]. 따라서 객체 인식 응용과 동시에 수행되는 메모리 집중적인 응용 프로그램의 수가 많으면 그림의 버퍼에서 병목 현상이 발생하여 성능이 저하될 수밖에 없다.

메모리 사용이 많은 태스크들이 동시에 수행되면 캐시 미스 횟수가 늘어날 확률이 높아지고, 이는 앞서 설명한 버퍼에서의 병목 현상과 더해져서 태스크들의 백엔드 스톨(Backend Stall) 사이클 숫자 증가로 나타난다[11]. 이는 CPU에 태스크를 할당하는 운영체제(리눅스 커널)의 스케줄러가 태스크를 할당 시 고려하지 않는 부분이다. 스케줄러는 시 분할 방식으로 태스크들에게 CPU 점유시간을 부여한다. 이때 백엔드 스톨 사이클이 많으면 부여받은 CPU 점유시간 중 상당 시간을 연산이나 메모리 접근을 하지 못한 채 기다리게 되어 해당 태스크의 수행시간이 늘어난다.

따라서 위에서 설명한 시스템 서버에서 사용되는 태스크들과 객체 인식 응용의 메인 태스크가 생성하는 워커 태스크들이 이러한 수행시간 지연을 겪게 되면 객체 인식 응용의 성능은 저하된다. 본 연구에서는 객체 인식 응용을 구성하는 태스크들이 시작된 후 백엔드 스톨시간 증가가 감지되면 해당 태스크들의 CPU 점유 시간을 늘린다. 이에 따라 CPU를 포함한 GPU, 메모리와 같은 시스템 자원을 사용할 수 있는 시간이 늘어나 성능 저하를 방지할 수 있다.

Ⅲ. 제안 방법

본 절에서는 객체 인식 과정과 지연을 줄 수 있는 요소들을 분석한 결과를 기반으로 솔루션을 제시한다. 구체적으로 시스템을 모델링하고 이후 솔루션의 구조와 세부 기법들을 기술적으로 설명한다.

1. 대상 시스템 모델링

그림 1에 나타낸 것처럼 대상 시스템의 소프트웨어는 응용과 라이브러리 계층과 리눅스 커널 계층 내부의 여러 태스크들로 이루어져 있다. 이 태스크들을 그룹화한 후 정의하면 시스템 서버, 커널 태스크, 응용 이렇게 세 가지로 구분할 수 있다.

시스템 서버 그룹은 앞서 설명한 디바이스 매니저, 윈도우 매니저, 디스플레이 서버를 포함하고 다른 응용에 기본적으로 필요한 서버 역할을 하는 태스크들로 이루어지며 $S=\{S_1,S_2,\cdots,S_m\}$로 나타낼 수 있다. 이때 디바이스 매니저, 윈도우 매니저, 디스플레이 서버를 구성하는 태스크들로 이루어진 그룹을 특별히 $S^o=\{S_i,\cdots,S_j\}$로 나타내고 $1\le i\le j\le m$을 만족하며 동시에 $S^o\subset S$ 관계를 만족한다. S^o는 객체 인식 응용뿐만 아니라 이를 필요로 하는 모든 응용에서 공통으로 사용한다. 커널 태스크 그룹은 응용 프로그램들과 가상 주소를 공유하지 않고 커널 단독으로 데몬이나 SoftIRQ[12] 등을 수행하기 위한 태스크들로 구성되며 CPU를 장시간 점유하지 않고 짧은 시간 동안 요구된 이벤트를 처리한다. 이는 $K=\{K_1,K_2,\cdots\}$로 표시한다. 마지막으로 응용 그룹은 객체 인식 응용을 포함하는 r개의 일반적인 응용프로그램들로 구성되며 $A=\{A^1\cup A^2\cdots\cup A^r\}$로 나타낸다. 이 중 객체 인식 응용 프로그램은 $A^o=\{M^o,w_1^o,w_2^o,\cdots,w_l^o\}$로 정의하며 $1\le o\le r$을 만족한다. $M^o\in A^o$를 만족하는 M^o는 해당 A^o의 메인 태스크를 나타내며 w_1^o, w_2^o,$\cdots$, w_l^o 는 메인 태스크가 생성하는 l개의 워커 태스크들을 나타낸다.

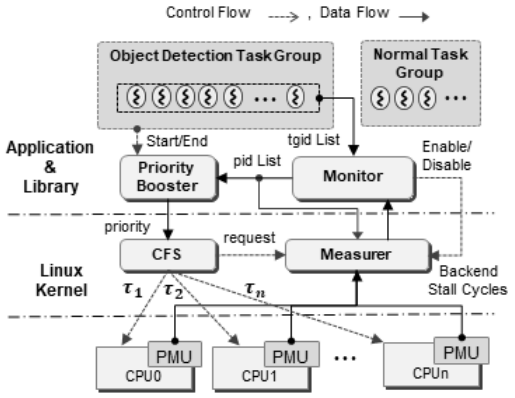

그림 3. 솔루션의 구조
Fig. 3. Solution architecture.

2. 솔루션의 구조

그림 3은 전체 솔루션의 구성을 나타낸다. 먼저 제안된 솔루션은 앞서 모델링에서 정의된 S, K, A 그룹을 객체 인식 응용에 필요한 태스크들의 그룹과 그 밖의 태스크들의 그룹으로 재구성한다. 이때 객체 인식 응용에 사용되는 태스크들은 $\{A^o \cup S^o\}$로 형성된다.

제안된 솔루션의 핵심은 $\tau_i \in \{A^o \cup S^o\}$을 만족하는 τ_i와 같은 태스크들에게 시스템 자원을 더 많은 시간 동안 사용할 수 있게 함이다. 앞 절에서 설명한 전체 객체 인식 응용의 과정을 보면 S^o에 속한 태스크들 역시 객체 인식 응용의 전체 수행 시간에 영향을 주기 때문에 무시할 수 없다. 따라서 A^o그룹과 더불어 제안된 솔루션을 적용 받게 한다.

응용 프로그램 수준에서 보면, 먼저 *Monitor*는 τ_i와 같은 태스크들을 찾아내고 이 결과를 *Priority Booster*에게 전달한다. 그 후 *Priority Booster*는 해당 태스크들의 우선순위를 변경한다. 리눅스 커널 수준에서는 *Measurer*를 두어 τ_i와 같은 태스크들의 성능 간섭 정도를 파악하고 *CFS* (Completely Fair Scheduler)[13]는 변경된 우선순위를 기반으로 스케줄링 정책을 변경하여 태스크들의 CPU 점유시간을 기존과 다르게 부여한다.

3. CFS (Completely Fair Scheduler)

본 연구에서 사용하는 태스크 스케줄러는 리눅스 커널의 CFS이다. 이는 VR(Virtual Runtime)을 기반으로 각 태스크들이 CPU를 점유하는 순서를 결정하며 어떤 태스크 τ_i의 VR은 다음의 식으로 나타낸다[13].

$$V_i(t) = \frac{1024}{W_i} \times C_i(t) \quad (1)$$

위 식의 $C_i(t)$는 τ_i가 생성된 후 시간 t 동안 CPU를 점유한 시간이며 W_i는 τ_i의 weight 값을 나타낸다. VR이 작을수록 CPU를 점유할 확률이 높아지기 때문에 weight 값이 클수록 같은 $C_i(t)$를 갖더라도 더 높은 확률로 CPU를 점유하게 된다.

CFS는 각 CPU마다 하나의 런큐를 할당한다. 런큐는 해당 CPU에서 수행 가능한 태스크들의 집합을 나타낸다. 이 런큐에 속한 τ_i가 CPU를 점유할 수 있는 타임 슬라이스는 다음과 같은 관계를 만족한다[13].

$$T_i \propto \frac{W_i}{\sum_{\tau_j \in T} W_j} \quad (2)$$

위 식 (2)의 T는 τ_i가 현재 속한 런큐를 나타낸다. 식에서 알 수 있듯이 weight 값이 클수록 큰 타임 슬라이스를 나타내어 더 장시간 CPU를 점유할 수 있게 된다.

4. 솔루션의 동작 방법

객체 인식 응용이 시작되면 이 응용의 메인 태스크인 M^o는 *Priority Booster*에게 응용의 시작(Start) 신호를 전달하여 시작을 알린다. M^o의 *pid*는 메인 태스크이기 때문에 *tgid* (Thread Group ID)를 나타낸다. 즉, 자신의 워커 태스크들인 w_1^o, w_2^o,…, w_l^o 의 부모 프로세스이다. M^o의 pid 인 *tgid*를 기반으로 Monitor 블록은 w_1^o, w_2^o,…, w_l^o의 *pid*들을 검출한다. 검출된 결과 $\{A^o \cup S^o\}$는 pid *List*로 형성되어 리눅스 커널의 *Measurer*에게 전달하고 *Enable* 신호를 통하여 *Measurer* 블록을 동작시킨다.

이후 *Measurer* 블록은 $\tau_i \in \{A^o \cup S^o\}$를 만족하는 τ_i가 어떤 CPU를 사용하게 되면 그 CPU 내부의 PMU (Performance Measurement Unit) 정보를 읽어서 백엔드 스톨 시간을 측정한다. 측정한 값을 τ_i가 CPU를 점유했던 전체 시간으로 나누어 그 비율이 어떤 한계점을 넘으면, *Monitor* 블록은 *Priority Booster*에게 *pid List*인 $\{A^o \cup S^o\}$를 전달하고 *Priority Booster*는 $\tau_i \in \{A^o \cup S^o\}$를 만족하는 τ_i의 nice 값을 변경하여 새로운 *priority* 값을 부여하고 이를 CFS에게 전달한다. 리눅스 커널에서 nice 값은 -20~19 사이의 값으로 정의 되며, -20이 가장 높은 *priority*를 나타내고 19가 가장 낮다.

CFS는 전달된 *priority* 값을 통하여 식 (2)의 W_i를 큰 값으로 변경한다. 이를 기반으로 τ_i의 식 (2)에 나타낸

타임 슬라이스 값은 증가하여 τ_i는 더 장시간 CPU를 점유하게 된다. 결과적으로 객체 인식 응용에 사용되는 태스크들의 CPU 점유시간이 늘어나고 그만큼 메모리 접근 혹은 GPU에게 명령어와 데이터를 전송할 수 있는 시간을 더 많이 확보함에 따라 성능을 향상시킬 수 있다.

시스템이 겪는 메모리로 인한 간섭 정도는 *Measurer* 블록이 *PMU*를 통하여 읽어오는 백엔드 스톨 시간으로 파악하였다. 이때 읽어오는 시점은 CFS가 사용하는 스케줄링 Tick을 기준으로 하고 이때 기준이 되는 시점이 되면 CFS는 *request* 신호를 발생시켜 *Measurer*가 측정하도록 만든다. 객체 인식 응용이 종료되면 *Priority Booster*는 해당 응용의 End 신호를 받게 된다. 그 후 *Priority Booster*는 시스템 호출을 통하여 CFS가 더 이상 request 신호를 발생시키지 않도록 한다. 이후 *Monitor* 블록은 *Disable* 신호를 발생시켜 *Measurer* 블록의 동작을 멈추게 한다.

Ⅳ. 실 험

본 절에서는 제안된 기법의 효용성을 입증하기 위한 실험 내용을 기술한다. 먼저 대상 시스템에 사용된 하드웨어와 소프트웨어를 소개하고 이어서 실험의 결과와 분석 내용을 설명한다.

1. 실험 대상 시스템

실험 대상 시스템으로 NVIDIA(사)의 Jetson AGX Xavier를 사용한다[14]. 이는 대표적인 자율주행 자동차에 사용될 수 있는 객체 인식용 임베디드 시스템이다[14]. 표 1은 이의 구체적인 사양을 보여준다.

실험에 사용된 응용은 세 가지로 나눌 수 있다. 먼저 시스템 메모리의 힙영역(malloc으로 할당)에 어떤 값을 집중적으로 읽고 쓰는 동작을 하는 메모리 집중적인 태스크, 두 번째 응용은 SPEC CPU2017[15], 마지막으로 DNN을 사용하는 객체 인식 응용인 YOLOv3-tiny[16]를 사용하고 임의의 도로를 찍은 동영상에서 자동차를 인식하게 하였다.

YOLOv3-tiny는 프레임마다 자동차들을 인식하고 그 결과를 테두리와 문자로 나타낸다. 이 응용의 성능은 FPS(Frame Per Second)와 수행시간 (End-to-End Latency)으로 나타내며 FPS가 클수록 많은 프레임에서 자동차 인식을 성공했다는 뜻이다.

표 1. Jetson AGX Xavier 사양
Table1. Jetson AGX Xavier specifications.

CPU	8-Core ARM v8.2 64-Bit CPU, 8MB L2 + 4MB L3
GPU	512-Core Volta GPU/64 Tensor Cores 11 TFLOPS (FP16), 22 TOPS (INT8)
Memory	16GB 256-bit LPDDR4x 2133MHz - 137GB/s
Storage	32GB eMMC 5.1
Linux kernel version	4.9.108

2. 실험 결과 및 분석

가. 메모리 경합과 백엔드 스톨 비율

첫 번째 실험으로 메모리 집중적인 태스크들이 동시에 수행될 때 그 숫자에 따라서 각자 겪게 되는 백엔드 스톨 비율을 측정하였다. 메모리 집중적인 태스크로는 SPEC CPU2017의 *lbm_s*를 사용하였다. 그림 4의 가로 축은 함께 수행되는 *lbm_s* 태스크의 숫자를 나타내고 세로축은 *lbm_s* 태스크들의 평균 백엔드 스톨 비율을 뜻한다. 그림에서 알 수 있듯이 8개가 동시에 수행될 경우 CPU 점유 시간의 반 이상(52.68%)의 시간동안 연산을 수행하지 못하고 기다리게 됨을 알 수 있다.

나. 메모리 집중 태스크들과 객체 인식 성능

다음 실험으로 시스템의 힙영역을 집중적으로 접근하는 태스크를 생성하여 메모리 사용의 경합을 만들고 그들의 숫자를 증가시키면서 YOLOv3-tiny의 성능을 측정하였다. 그림 5의 가로 축은 메모리 집중 태스크들의 숫자를 나타내며, 그림의 (a)는 DNN 연산을 GPU를 사용하는 경우이고 (b)는 CPU를 사용한 경우이다. 그림의 "*Org*"는 솔루션을 적용하지 않은 경우이고 "*Solution*"은 적용한 경우를 나타낸다. 그림에서 볼 수 있듯이 최대 33% 객체 인식 성능이 향상됨을 알 수 있다. 추가적으로 GPU를 사용한 DNN 연산을 수행 시 객체 인식 성능이 CPU를 사용한 경우보다 20배 이상 높음을 알 수 있다.

다. SPEC CPU2017과 객체 인식 성능

마지막 실험으로 SPEC CPU2017 벤치마크 프로그램 중 *lbm_s*, *imagick_s*, *xalancbmk_s*, *omnetpp_s*를 선택한 후 코어 개수인 8개만큼 각각 생성하여 YOLOv3-tiny와 동시

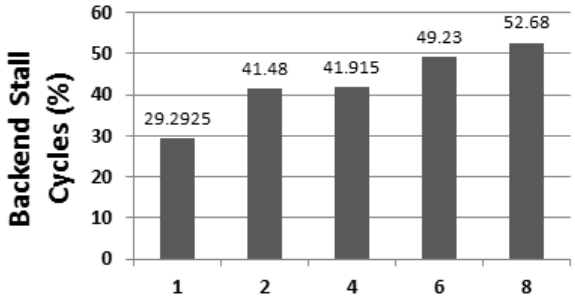

그림 4. 메모리 집중 태스크 숫자에 따른 백엔드 스톨 비율
Fig. 4. Backend stall cycle ratio according to the number of memory intensive tasks.

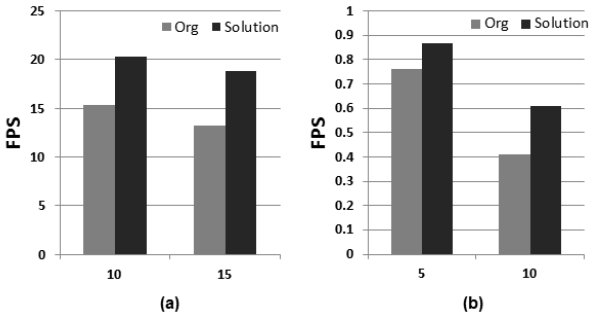

그림 5. YOLOv3-tiny의 메모리 집중 태스크 숫자에 따른 성능: (a) GPU로 DNN 연산, (b) CPU로 DNN 연산
Fig. 5. Performance of YOLOv3-tiny according to the number of synthetic memory intensive tasks: when DNN is computed (a) by GPU, and (b) by CPU.

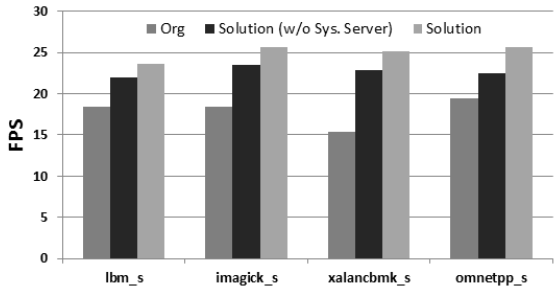

그림 6. YOLOv3-tiny의 SPEC CPU17 벤치마크 프로그램과 동시 수행 시 FPS 성능
Fig. 6. FPS Performance of YOLOv3-tiny running with SPEC CPU17 benchmark programs.

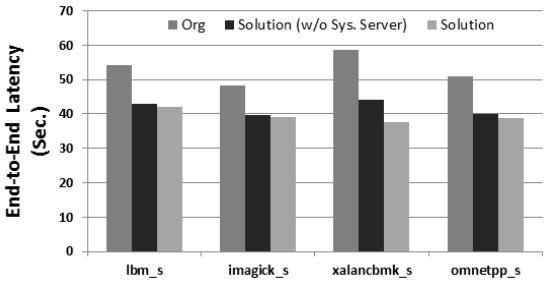

그림 7. YOLOv3-tiny의 SPEC CPU17 벤치마크 프로그램과 동시 수행 시 총 수행시간
Fig. 7. End-to-End latency of YOLOv3-tiny running with SPEC CPU17 benchmark programs.

에 수행하면서 성능을 측정하였다. 그림 6은 이때 측정한 FPS 성능을 나타낸다. 그림 1과 Ⅲ절의 시스템 모델링을 통하여 알 수 있듯이 전체 객체 인식 과정은 $\{A^o \cup S^o\}$에 속한 태스크들로 구성된다. 본 실험에서는 S^o가 전체 객체 인식 성능에 영향을 미치는 정도도 함께 알아본다. 그림에서 "*Solution (w/o Sys. Server)*"로 표기된 데이터는 A^o만 제안된 기법을 적용하고 S^o는 제외했을 때의 결과를 나타낸다.

그림 6에서 알 수 있듯이 모든 경우에 있어서 성능 향상을 나타내었고 최소 향상은 *omnetpp_s* 과 수행될 때의 경우이며 약 13.5%, 최대 성능 향상은 *xalancbmk_s* 와의 경우이며 약 33%를 기록하였다. 다음으로 동일 조건에서 총 수행시간 즉, 전체 응용의 지연 시간을 측정하고 그림 7에 나타내었다. FPS 결과와 마찬가지로 *xalancbmk_s* 와 동시 수행의 경우 최대 약 25% 수행시간 감소를 나타내었다.

그림 6과 그림 7에서 알 수 있듯이 시스템 서버 태스크 그룹인 S^o까지 적용된 기법을 적용 시 A^o만 제안된 기법을 적용 때보다 최대 FPS는 12.5%, 수행시간은 15% 향상되었다. 이를 통하여 객체 인식의 성능에 있어서 GPU를 사용하여 DNN 연산을 수행하는 객체 인식 응용의 메인 태스크와 이의 워커 태스크들(A^o)뿐만 아니라 이들에게 도움을 주는 시스템 서버들에 속한 태스크들(S^o)의 성능도 전체 응용의 성능에 영향을 끼친다는 것을 알 수 있다.

Ⅴ. 결 론

자율주행 자동차에 있어서 객체 인식 기능은 안전에 관련된 최상위 중요도를 갖는 응용 프로그램이다. 최근 딥러닝 기술의 발전으로 객체 인식은 인공 신경망 즉, DNN을 사용하고 있으며 이는 큰 규모의 산술연산과 메모리를 사용하기 때문에 다른 응용들과 동시에 수행될 때 성능 저하가 발생할 수 있고, 이는 안전에 위협을 줄 수 있다. 본 논문에서는 메모리 간섭으로 인한 성능 저하 요인을 분석과 실험을 통하여 보였다. 이를 해결하기 위하여 리눅스 커널 수준에서 객체 인식이 시작되면 시스템 자원을 더 사용할 수 있게 하고 종료 시 원

래의 상태로 복귀할 수 있는 솔루션을 개발하였다. YOLOv3-tiny를 객체 인식 응용으로 사용하여 실험한 결과 SPEC CPU2017 벤치마크 프로그램들에게 성능 간섭을 받게 하였을 때, 솔루션 적용 전보다 FPS는 최대 33% 증가 하였고 지연 시간은 25% 감소하였다.

REFERENCES

[1] E. Rolf, and M. D. Natale, “Mixed Criticality Systems－A History of Misconceptions?”, IEEE Design & Test Vol. 33, No. 5, pp. 65-74, 2016.
[2] P. Fekrl, M. Zadeh, and J. Dargahi, “On the Safety of Autonomous Driving: A Dynamic Deep Object Detection Approach”, SAE Technical Paper 2019-01-1044, 2019
[3] J. Kim, P. Shin, S. Noh, D. Ham and S. Hong, “Reducing Memory Interference Latency of Safety-Critical Applications via Memory Request Throttling and Linux Cgroup”, in Proceedings of the 31st IEEE Conference on International System on chip, 2018.
[4] V. Sze, Y. Chen, T. Yang, and J. S. Emer, “Efficient Processing of Deep Neural Networks: A Tutorial and Survey,” in Proceedings of the IEEE, Vol. 105, no. 12, pp. 2295-2329, Jan. 2017.
[5] J. Lee, S. Lee and S. Yang, “Model Ensemble for Speed Enhancement in Object Detection”, Journal of the Institute of Electronics and Information Engineers, Vol. 56, No. 6, June 2019.
[6] J. Choi, D. Hwang, J. An and J. Lee, “Object Detection Using CNN for Automatic Landing of Drones”, Journal of the Institute of Electronics and Information Engineers, Vol. 56, No. 5, May 2019.
[7] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” in Proceeding of the International Conference on Learning Representations, San Diego, CA, 2015.
[8] https://developer.nvidia.com/cudnn
[9] Y. Chen, T. Krishna, J. S. Emer, V. Sze, “Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks,” IEEE Journal of Solid-State Circuits, Vol. 52, no. 1, pp. 127-138, Nov. 2017.
[10] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, “DaDianNao: A machine-learning supercomputer,” in Proceeding of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, Cambridge, UK, pp. 609-622, 2014.
[11] A. Yasin, “A Top-Down method for performance analysis and counters architecture”, Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, pp. 35-44, 2014
[12] https://lwn.net/Articles/520076/
[13] S. Huh, J. Yoo, M. Kim and S. Hong, “Providing Fair Share Scheduling on Multicore Cloud Servers via Virtual Runtime-based Task Migration Algorithm”, Proceedings of the 32nd IEEE International Conference on Distributed Computing Systems (ICDCS), pp. 606-614, Jun 2012.
[14] https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit
[15] A. Limaye and T. Adegbija, “A Workload Characterization of the SPEC CPU2017 Benchmark Suite,” IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Belfast, 2018, pp. 149-158.
[16] R. Joseph, S. K. Divvala, R. B. Girshick, A. Farhadi, “You only look once: Unified, real-time object detection”, Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

저 자 소 개

김 명 선(정회원)
2000년～2002년 LG전자 전송연구소 주임연구원
2002년～2011년 삼성전자 DMC 연구소 책임연구원
2016년 서울대학교 전기컴퓨터공학부 박사 졸업.
2016년～2019년 삼성전자 SR연구소 수석연구원
2019년～현재 한성대학교 IT융합공학부 조교수
<주관심분야 : 인공지능 가속기, Linux kernel, HW/SW Co-desgin 등>