Journal of The Institute of Electronics and Information Engineers Vol.58, NO.10, October 2021 https://doi.org/10.5573/ieie.2021.58.10.27
ISSN 2287-5026(Print) / ISSN 2288-159X(Online)

논문 2021-58-10-4

오버헤드 최소화를 통한 임베디드 GPU 환경에서 멀티 DNN 추론 성능 향상

(Improving Multi-DNN Inference Performance under Embedded GPU Environments with Overhead Minimization)

임 철 순*, 김 명 선**

(Cheolsun Lim and Myungsun Kim©)

요 약

기존 임베디드 시스템에서는 DNN 연산을 전적으로 서버 시스템에 의존했었지만 임베디드 GPU의 성능 향상으로 임베디드 시스템 자체적으로 DNN 연산을 수행할 수 있게 되었다. 따라서 클라우드에 의존해서 DNN 연산을 수행하던 무인항공기, 스마트 시티, 자율주행 자동차 등의 시스템들이 DNN을 시스템 내부 자원만으로 실행하는 능력이 생겼다. 특히 자율주행 자동차의 경우 여러 개의 카메라와 센서들로부터 입력되는 다양한 데이터는 영상인식 및 차선 감지 등의 응용에 전달된다. 이후 각각의 응용 들은 여러 종류의 DNN을 사용하여 이 데이터를 처리한다. 여러 개의 DNN을 사용하기 위해서는 각 DNN을 프로세스 또는 쓰레드에 할당하여 일괄적으로 실행해야 하고 실행하는 환경에 따라 전체 실행시간에 많은 차이가 있다. 본 논문에서는 첫 번째로 멀티 컨텍스트 환경과 싱글 컨텍스트 환경에서 여러 개의 DNN을 실행될 때 각 환경에서 발생하는 오버헤드를 분석한다. 두 번째로 임베디드 GPU 시스템에서 발생할 수 있는 CPU와 GPU 사이의 메모리 복사 문제와 이와 관련된 캐시 이슈에 대하여 분석한다. 마지막으로 분석된 문제를 해결하고 오버헤드를 최소화 할 수 있는 프레임워크를 제안한다. 제안된 프레임워크를 상용 보드에 적용 후 실험한 결과 멀티 컨텍스트 환경 대비 최대 40.8%의 실행시간이 감소하였다.

Abstract

In traditional embedded systems, DNN operations were entirely dependent on server systems, but improved performance of embedded GPUs allowed the embedded system to perform DNN operations on its own. Thus, systems such as unmanned aerial vehicles, smart cities, and self-driving cars, which relied on the cloud to perform DNN operations, have the ability to run DNN models only with internal resources. Especially for self-driving cars, various data input from multiple cameras and sensors is transmitted to applications such as image recognition and lane detection. Subsequently, each application uses several kinds of DNNs to process the data. To use multiple DNNs, each DNN must be allocated to a process or a thread to run in batches, and there are many differences in overall execution time depending on the environment in which it runs. In this paper, we first analyze the overhead that occurs in each environment when multiple DNNs are executed in a multi-context environment and a single-context environment. Secondly, we analyze the memory copy problem between CPU and GPU that can occur in embedded GPU systems and the cache issue associated with it. Finally, we propose a framework that can solve the analyzed problem and minimize overhead. After applying the proposed framework to a commercial board, experiments result in up to 40.8% reduction in running time compared to the multi-context environment.

Keywords : Deep learning, GPU, Multi-DNN execution, Embedded system

*학생회원, **정회원, 한성대학교 IT융합공학부(Department of IT Convergence Engineering, Hansung University)
© Corresponding Author(E-mail : kmsjames@hansung.ac.kr)
※ 본 연구는 한성대학교 교내학술연구비 지원과제임.
Received ; July 7, 2021 Revised ; August 6, 2021 Accepted ; September 7, 2021

Ⅰ. 서 론

최근 DNN의 발전으로 여러 분야에서 DNN을 핵심 기술로 사용하여 자신들의 애플리케이션에 적용하려는 시도가 늘고 있다. DNN의 학습과 추론은 매우 큰 데이터를 요구하고 이를 효과적으로 처리해야 하므로 보통 우수한 병렬처리 특성을 갖춘 GPU 기반 서버 시스템을 이용한다. 하지만 네트워크 지연과 개인 정보보호 이슈로 인하여 서버 시스템보다는 임베디드 시스템 내부에서 DNN을 직접 처리하는 경우가 보편화되고 있다.

이러한 트렌드를 반영하여 GPU 기반 임베디드 시스템이 크게 발전했고 내장된 고성능 GPU를 이용하여 DNN을 빠른 시간 내에 실행할 수 있게 되었다. 이를 응용하여 드론과 같은 무인항공기에 GPU 기반 임베디드 시스템을 장착하여 운용 중 드론의 자체 시스템을 이용하여 객체 탐지를 할 수 있게 되었고[1], 스마트 시티의 분야에서도 CCTV자체에서 객체 탐지 및 분석을 할 수 있도록 연구가 되고 있다[2].

이런 분야들에 더하여 GPU 기반 임베디드 시스템을 자율주행 자동차 분야에서 사용하기 위하여 많은 연구가 이루어지고 있다. 특히 NVIDIA(사)는 자율주행자동차를 위한 임베디드 시스템 및 개발환경을 적극적으로 지원하고 있다[3]. 자율주행 자동차 내부의 많은 응용들은 여러 개의 카메라와 센서에서 들어온 데이터를 여러 종류의 DNN 모델들을 사용하여 실시간으로 처리한다. 다시 말하면 임의의 한 시점에서 여러 개의 DNN 응용 프로그램들이 임베디드 시스템내부에서 실행된다고 할 수 있다.

서버 컴퓨팅 환경 대비 시스템 리소스가 제한된 임베디드 시스템에서는 여러 개의 DNN 모델들이 동시에 처리될 때 여러 가지 문제점이 발생한다. 여러 응용프로그램이 저마다 별개의 프로세스(멀티 컨텍스트)로 수행될 경우 컨텍스트 교환 오버헤드가 있고, 이와 반대로 한 프로세스 내부에 여러 DNN 모델을 저마다의 쓰레드(싱글 컨텍스트)로 생성하는 경우에는 GPU를 공유자원으로 정의하고 쓰레드마다 동기화 기법의 제어를 받게 되어 오버헤드가 발생한다. 이때 발생하는 오버헤드로 인해 여러 개의 DNN 응용 프로그램을 실행할 때 각 환경에 따라 추론 속도에 차이가 생긴다.

따라서 본 논문에서는 이러한 문제점을 해결하기 위하여 먼저 멀티/싱글 컨텍스트 각 환경에서의 오버헤드를 분석한다. 분석된 결과를 바탕으로 여러 개의 DNN 응용 프로그램이 임베디드 GPU를 공유한 형태로 실행될 때 멀티/싱글 컨텍스트 환경의 오버헤드를 최소화할 수 있는 프레임워크를 제안한다.

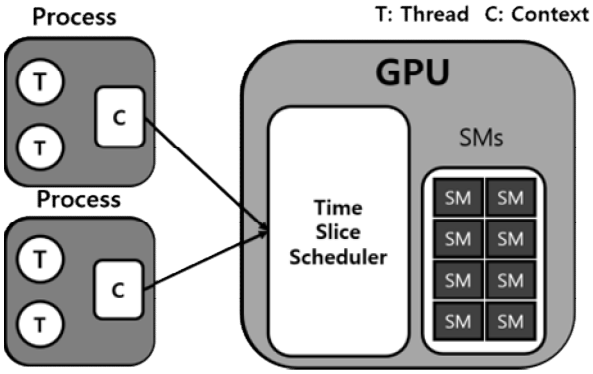

그림 1. GPU를 사용하는 CPU 프로세스의 구조
Fig. 1. Structure of CPU processes using GPU.

Ⅱ. 대상 시스템

1. 임베디드 GPU 기반 시스템

일반적으로 하나의 GPU는 여러 개의 코어로 구성되고 CPU는 단일 코어 기반으로 구성된다. GPU의 코어들은 CPU의 코어보다 낮은 성능을 보이지만 훨씬 많은 코어를 가지고 있기 때문에 대규모 병렬처리에 GPU가 더 적합하다. 그래서 대규모 병렬처리를 필요로 하는 DNN 연산은 주로 GPU를 사용한다. 일반적인 GPU 기반 시스템은 CPU, 시스템 메모리, GPU, 디바이스 메모리로 구성된다. 시스템 메모리는 CPU에서 사용하는 메모리 공간이고 디바이스 메모리는 GPU에서 사용하는 메모리 공간이다. GPU는 시스템 메모리에 존재하는 값들을 바로 사용할 수 없고 GPU의 디바이스 메모리로 전달한 후에 GPU에서 사용할 수 있다.

본 논문에서는 임베디드 GPU 기반 시스템인 Jetson AGX Xavier(이하 Xavier)를[4] 타겟 디바이스로 사용한다. Xavier는 System-on-Chip(SoC) 방식으로 설계되어 소비전력이 적고 소형화가 가능한 특징이 있지만 서버용 GPU에 비해 낮은 성능을 보인다. 임베디드 시스템이기 때문에 GPU의 디바이스 메모리가 별도로 존재하지 않고 CPU가 사용하는 시스템 메모리의 일부를 GPU가 사용하는 디바이스 메모리로 사용한다.

2. GPU의 동작 방식

그림 1은 GPU 기반 시스템의 구조를 나타낸다. CPU의 프로세스는 GPU에서 컨텍스트라는 단위로 동작하게 된다. 각 프로세스 내의 쓰레드에서 GPU 작업들은

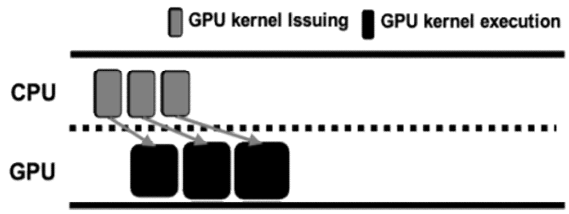

그림 2. GPU 커널함수의 동작 방식
Fig. 2. Execution method of GPU kernel function.

컨텍스트를 통해 GPU로 전달된다. GPU에서는 한 시점에 하나의 컨텍스트만 활성화될 수 있으며 런타임 동안 여러 컨텍스트가 존재할 때에는 GPU내부의 그림 1의 Time Slice Scheduler를 통해 각 컨텍스트에 존재하는 GPU 작업들을 번갈아 가며 실행한다[5].

그림 2는 커널함수의 동작을 나타낸다. GPU는 CPU를 통해 명령을 받아야만 동작을 하며 이때 GPU에서 실행되는 GPU 작업을 커널함수라 한다. 커널함수를 실행하는 단계는 다음과 같다. ① CPU에서 GPU로 메모리 값을 복사한다. ② CPU에서 GPU로 커널함수의 실행 정보를 보낸다. ③ GPU는 CPU에서 받아온 정보를 토대로 커널함수를 실행한다. ④ CPU에서 값을 요청하면 GPU의 메모리 값을 CPU로 복사한다. 이 때 커널함수는 들어온 순서대로 실행되며 이는 CPU와 비동기 방식으로 실행된다. CPU에서는 GPU에서 먼저 실행한 커널함수의 종료여부와 상관없이 다음 커널함수가 있다면 다음 커널함수를 발행한다.

3. 소프트웨어 지원

NVIDIA(사)에서는 응용프로그램 수준에서 GPU를 사용할 수 있도록 CUDA[6]를 지원한다. CUDA는 저수준 API인 드라이버 API와[7] 고수준 API인 런타임 API를[8] 모두 지원한다. 추가로 딥러닝 가속을 위한 cuDNN[9], 선형대수학을 위한 cuBLAS[10] 등의 라이브러리를 제공하고 있다. 런타임 API와 드라이버 API를 사용한 응용 프로그램은 GPU에서의 작업을 위해 사용할 자원인 쓰레드와 블럭을 직접 설정해야 한다. 쓰레드와 블록을 직접 설정하기 위해서는 커널함수의 입력값과 출력값, 사용하는 GPU의 특성을 알아야 하며 적절하게 설정해 주어야 GPU를 효율적으로 이용할 수 있다. 반면에 cuDNN, cuBLAS를 이용한 응용프로그램은 작업의 특성에 맞게 효율적으로 쓰레드와 블록을 자동으로 설정해준다.

보통 커널함수를 실행할 때 GPU의 사용가능한 SM (streaming multiprocessor)이 남아도 한 번에 한 개의

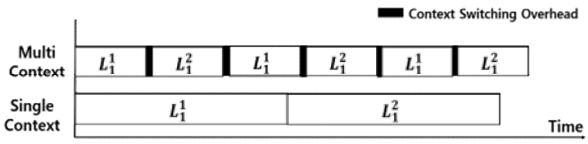

그림 3. 멀티컨텍스트와 싱글컨텍스트의 실행시간 비교
Fig. 3. Comparison of execution times between multi-context and single-context.

커널함수만 실행할 수 있지만 남는 자원을 사용하여 동시에 커널함수를 실행시킬 수 있는 CUDA 스트림 API를[11] 제공한다. 스트림을 이용하면 GPU의 SM들이 각 스트림에 존재하는 커널함수들을 동시에 실행할 수 있어 GPU의 병렬성을 극대화 할 수 있다.

Ⅲ. 멀티 DNN 추론 환경 및 문제점 분석

각 DNN은 여러 개의 연속된 레이어들로 구성되어 있고 각 레이어는 커널함수로 구현되어 있다. n번째 DNN인 DNN_n에 k개의 레이어가 존재하면 $DNN_n = \{L_1^n, L_2^n, \cdots, L_k^n\}$로 나타낼 수 있다.

본 논문에서는 여러 개의 DNN을 동시에 실행하는 것을 멀티 DNN 실행이라 정의한다. 그리고 멀티 DNN 실행 환경을 멀티 컨텍스트(각 프로세스 당 하나의 DNN)와 싱글 컨텍스트(한 프로세스 내부에 복수개의 DNN)로 구분하고 각 환경에서의 문제점을 분석한다.

1. 멀티 컨텍스트 환경

2장에서 말한 것과 같이 CUDA에서는 프로세스 컨텍스트라고 하며 모든 동작은 컨텍스트 단위로 동작한다. 한 컨텍스트는 여러 쓰레드를 가질 수 있고 프로세스는 여러 개의 컨텍스트를 가질 수 있지만 CUDA에서는 프로세스당 하나의 컨텍스트를 권장하고 있다[5]. 임의의 한 시점에 하나의 컨텍스트만 GPU를 점유할 수 있기 때문에 여러 컨텍스트가 활성화 되어있는 상태에서는 Time Slice Scheduler에 의하여 각 컨텍스트가 번갈아 가면서 GPU를 점유한다.

이때 컨텍스트가 바뀌면서 기존 컨텍스트에 담겨있는 정보들을 메모리에 저장하고 다음에 GPU를 점유할 컨텍스트의 정보를 가져오는 컨텍스트 교환을 하게 되는데 이로 인해 발생하는 오버헤드를 컨텍스트 교환 오버헤드라 한다.

그림 3에서는 DNN의 레이어 L_1^1과 L_1^2가 각각 멀티 컨텍스트 환경과 싱글 컨텍스트 환경에서 동작하는 것

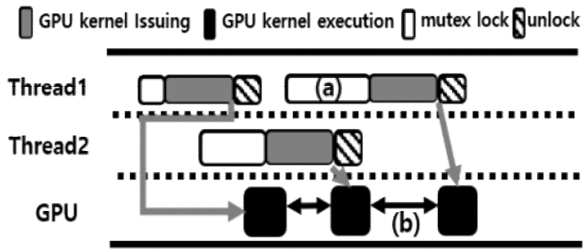

그림 4. 싱글 컨텍스트 환경에서의 mutex 오버헤드
Fig. 4. Overhead incurred by mutex in single context environment.

을 보여준다. 멀티 컨텍스트 환경의 경우 두 레이어가 실행되는 동안 일정한 시간마다 컨텍스트 교환이 일어난다. 반면에 싱글 컨텍스트 환경에서는 컨텍스트 교환 없이 각 레이어가 실행되므로 오버헤드가 없다. 그렇기 때문에 두 환경 모두 같은 레이어를 실행하지만 컨텍스트 교환 오버헤드로 인해 멀티 컨텍스트 환경이 더 많은 시간이 걸린다.

2. 싱글 컨텍스트 환경

DNN을 가장 이상적으로 실행하는 형태는 그림 2와 같이 이전에 발행된 커널함수가 끝나기 전에 다음 커널함수를 발행하여 GPU가 유휴시간 없이 동작하도록 하는 것이다. GPU의 커널함수 실행 자체는 CPU와 독립적이지만 CPU에서 스케줄링 되는 시점에 따라서 GPU에 유휴시간이 발생하게 된다. 유휴시간은 시스템 자원은 충분하지만 사용을 하지 못하는 상태이므로 이는 고스란히 프로그램 실행시간에 영향을 미치게 된다.

프로세스에 존재하는 여러 쓰레드는 하나의 컨텍스트를 공유할 수 있다. 멀티 DNN 실행을 위해 하나의 프로세스에 각 DNN당 쓰레드를 할당하여 실행하면 각 쓰레드들이 하나의 컨텍스트를 통하여 일제히 커널함수를 발행하게 된다. 그림 4는 각 쓰레드가 커널함수를 발행하는 동작을 보여주고 있다. 각 쓰레드가 GPU에 커널함수를 발행할 때 GPU를 공유자원으로 정의하고 동기화 함수(mutex)를 이용하여 여러 쓰레드가 공유자원에 동시에 접근하지 못하도록 제어한다.

따라서 여러 쓰레드가 공유 자원인 GPU에 동시 접근을 시도하면 각 쓰레드들은 mutex 획득/해제를 수행하면서 직렬화(serialize) 구간을 늘리게 된다. CPU에서 수행되는 함수가 mutex 획득/해제를 하는 동안 GPU가 사용가능함에도 불구하고 아무런 커널함수를 할당 할 수 없다. 그림 4는 mutex 오버헤드가 주는 영향을 보여주는데 mutex로 인해 (a)와 같이 CPU에서 커널 발행이 늦어지게 된다. GPU는 CPU에서 커널함수를 발행해야만 실행할 수 있기 때문에 (a)로 인하여 GPU에 (b)와 같이 유휴시간이 발생하는 것을 보여준다.

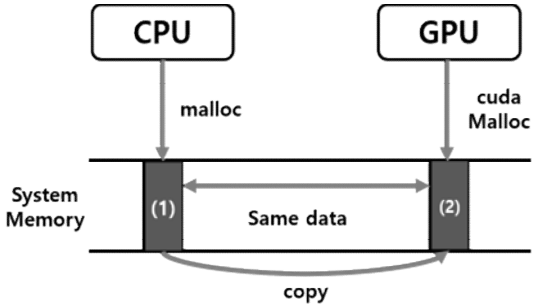

그림 5. CUDA API를 이용한 메모리 할당 및 복사
Fig. 5. Memory allocation and copy using CUDA API.

3. 데이터 복사 문제

일반적인 경우 DNN의 입력 데이터는 호스트인 CPU가 GPU에게 전달한다. 이러한 전달 과정에서는 GPU가 사용하는 메모리로 입력 데이터를 복사해 주어야 한다. 그림 5는 Xavier에서 CUDA를 이용하여 데이터를 복사하는 예시를 보여준다. 메모리 공간을 확보하기 위해 CPU에서는 malloc함수를 GPU에서는 cudaMalloc함수를 이용한다. 다음으로 CPU의 데이터를 GPU에서 사용하기 위해 cudaMemcpy함수를 이용하여 CPU의 데이터를 GPU로 복사한다.

서버 시스템의 경우 GPU는 별도의 디바이스 메모리를 사용하므로 그림 5의 (1)영역과 (2)영역이 서로 다른 메모리에 존재한다. 하지만 대상시스템인 Xavier는 CPU와 GPU가 시스템 메모리를 공유하므로 그림 5에서 보는 것처럼 동일 메모리 내부의 다른 영역에 존재한다. 따라서 복사를 할 경우 동일한 값이 같은 메모리 내부의 다른 영역에 존재하게 되어 매우 비효율적으로 메모리를 사용하고, 또한 필요 없는 데이터 복사로 인해 CPU 싸이클을 낭비하게 된다.

Ⅳ. 멀티 DNN 추론 성능 최적화 기법 제안

1. 오버헤드 최소화 기법 제안

본 절에서는 3장에서 언급한 두 가지 오버헤드를 최소화 할 수 있는 프레임워크를 제안한다. 그림 6은 본 논문에서 제안하는 프레임워크의 구조를 보여주고 있다. 첫 번째로 멀티 컨텍스트 환경에서 오는 컨텍스트 교환 오버헤드를 줄이기 위해 싱글 컨텍스트 환경을 기반으로 하나의 프로세스에 각 DNN 쓰레드를 (a)와 같

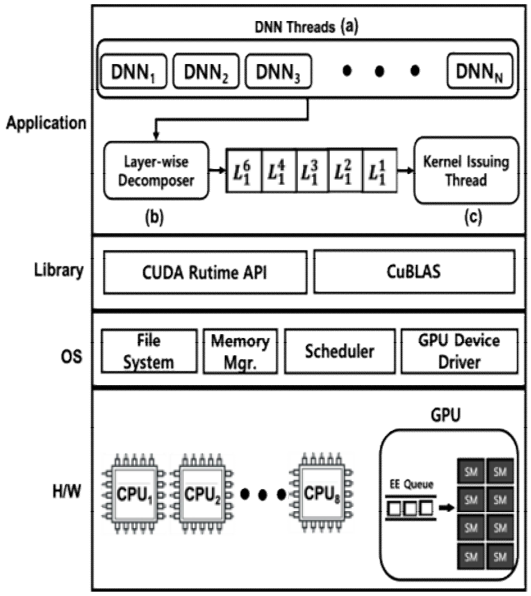

그림 6. 오버헤드 최소화된 멀티 DNN 실행 구조
Fig. 6. Overhead minimized Multi-DNN execution framework.

이 할당한다. 두 번째로 싱글 컨텍스트 환경에서 오는 mutex 오버헤드를 줄이기 위해 커널발행을 하는 쓰레드를 한 개만 사용한다. 커널발행을 담당하는 쓰레드(c)는 다른 행동은 하지 않고 DNN 쓰레드에서 오는 레이어들을 GPU에 커널함수로 발행하는 역할만 한다. 커널발행 쓰레드는 한 번에 하나의 커널함수만 발행할 수 있기 때문에 모든 DNN 쓰레드에서 오는 여러 레이어를 동시에 처리할 수 없다. 따라서 큐를 이용하여 DNN 쓰레드에서 보낸 레이어를 큐에서 대기시키고 커널 발행 쓰레드가 순서대로 처리 한다.

본 프레임워크의 작동순서는 다음과 같다. (a)의 DNN 쓰레드들이 레이어를 GPU로 직접 보내지 않고 Layer-wise Decomposer(b)를 통해 큐로 보낸다. (c)는 큐에 있는 레이어를 하나씩 꺼내 GPU로 커널함수를 발행한다. 발행된 커널함수는 GPU의 스트림에서 대기하며 스트림에서 제일 앞에 존재하는 커널함수를 EE Queue에 대기시킨다. 마지막으로 GPU는 EE Queue에 있는 커널함수를 하나씩 꺼내어 SM을 이용해서 실행한다[12]. 그림 6은 큐의 제일 앞에 존재하는 레이어인 L_1^1이 다음번에 (c)를 통해 커널이 GPU에 발행되는 예를 보여준다.

큐도 일종의 공유자원이므로 각 DNN 쓰레드가 큐에 접근하는 것을 제어하기 위해 mutex(이하 Qmutex)를 사용한다. DNN의 모든 레이어들을 하나씩 큐에 넣게 되면 레이어의 개수만큼 큐에 접근하고 접근할 때마다

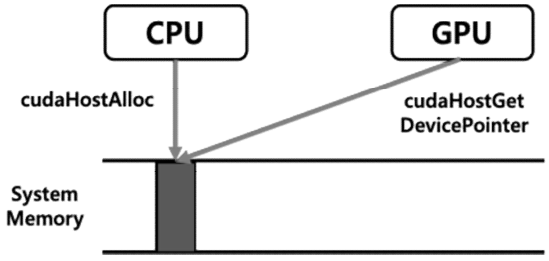

그림 7. Xavier의 CUDA 제로카피
Fig. 7. CUDA zero-copy in Xavier.

Qmutex에 대한 획득/해제를 해야 한다. (b)는 상대적으로 짧은 실행시간을 가진 Activation, Batch Normalization과 같은 레이어들을 Convolution 레이어와 하나의 구조체로 묶어서 큐에 전달한다. 따라서 큐에 접근하는 횟수를 줄일 수 있으므로 Qmutex에 대한 획득/해제의 횟수도 줄어들기 때문에 Qmutex로 인한 오버헤드가 줄어든다.

2. CUDA 제로카피 적용

3장에서 언급한 데이터 복사 문제를 CUDA에서 제공하는 제로카피 기법을 사용하여 해결한다[13]. 제로카피는 명시적으로 데이터를 복사하는 기존의 방법과 달리 하나의 메모리 공간을 CPU와 GPU가 같이 사용한다. 서버시스템의 경우 제로카피를 사용 시 커널함수 실행마다 PCIe 버스를 통해 데이터를 가져오는데 이는 어느 정도의 시간을 필요로 한다. 하지만 대상 시스템인 Xavier는 하나의 시스템 메모리를 공유하므로 GPU가 CPU쪽 데이터가 들어있는 메모리에 직접 접근이 가능하기 때문에 제로카피를 사용해도 데이터 접근에 소요되는 오버헤드가 없다.

그림 7은 제로카피의 예시를 보여준다. 제로카피 사용을 위해서는 페이징 메모리를 할당받는 malloc대신 cudaHostAlloc함수를 통해서 고정된 메모리 공간을 할당받아야 한다. 그 다음 GPU의 메모리 공간을 할당받지 않고 cudaHostGetDevicePointer함수를 통해 CPU와 같은 메모리 공간을 가르키게 한다. 기존의 복사방식에서는 하나의 데이터를 위해 두 개의 메모리 공간을 할당하였지만 제로카피를 이용하면 하나의 메모리 공간을 공유해서 사용할 수 있다.

CUDA 9.0 이전에는 CPU와 GPU가 제로카피로 사용되는 메모리 영역에 대해서는 캐싱을 하지 않기 때문에 데이터에 접근할 때 무조건 시스템 메모리를 통해 접근했다. CUDA 9.0 이후부터는 CPU쪽 캐시에서 제로카피 메모리 영역에 대하여 캐싱하는 기능을 지원하기 시작

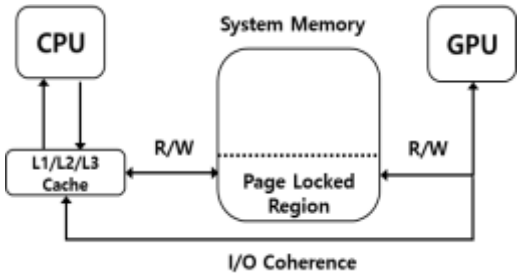

그림 8. Xavier에서 제공되는 CPU와 GPU 사이의 캐시동기화 기법[14]
Fig. 8. Cache synchronization technique between CPU and GPU provided by Xavier[14].

했지만 GPU에서는 제로카피 메모리에 대한 캐싱을 지원하지 않기 때문에 여전히 시스템 메모리에 대한 접근이 필요했다. 하지만 Xavier이후의 하드웨어에서는 그림 8과 같이 I/O Coherence[14]를 제공하며 GPU는 제로카피 메모리에 대한 캐싱이 불가능하지만 대신 CPU의 캐시메모리를 참조하여 빠른 데이터 접근이 가능하게 되었다.

V. 실 험

본 절에서는 멀티 DNN실행을 멀티 컨텍스트 환경과 싱글 컨텍스트 환경, 4장에서 제안한 프레임워크를 각각 비교하여 결과를 분석한다.

1. 실험 대상 시스템

본 실험은 NVIDIA(사)의 Jetson AGX Xavier를 이용하여 멀티 DNN실행의 결과를 분석하였다. Xavier는 다양한 분야의 오토노머스 머신 및 엣지 컴퓨팅을 위한 ARM 기반의 고성능 임베디드 시스템이다. 표 1은 Xavier의 하드웨어와 소프트웨어의 사양을 나타낸다.

본 실험에서는 멀티 컨텍스트 환경(이하 M-C), 싱글 컨텍스트 환경(이하 S-C)과 4장에서 제안한 프레임워크(이하 Q-S)과의 비교를 통하여 총 실행시간이 얼마나 개선되었는지를 확인한다.

표 1. NVIDIA(사) Jetson AGX Xavier 구성
Table 1. Jetson AGX Xavier specifications.

GPU	512-Core Volta GPU/64 Tensor Cores 11 TFLOPS (FP16), 22 TOPS (INT8)
CPU	8-Core ARM v8.2 64-Bit CPU, 8MB L2 + 4MB L3
Memory	32GB 256-bit LPDDR4x 2133MHz - 137GB/s
OS	Linux kernel version 4.9.140-tegra
JetPack	v.4.2 [L4T 32.1.0]
CUDA ver.	CUDA 10.0.166

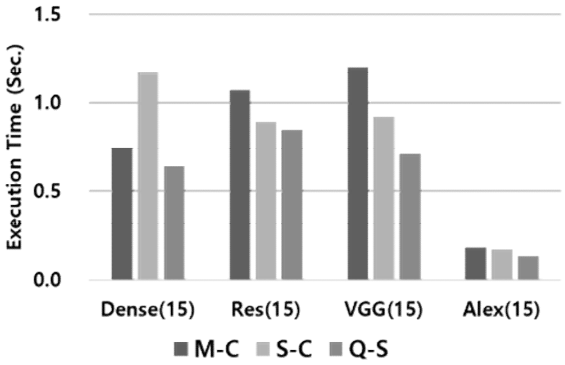

그림 9. 실행 환경에 따른 동일 종류 DNN의 실행 시간
Fig. 9. Execution times for DNNs of the same type according to the execution environment.

실험에서는 DenseNet201[15], ResNet152[16], VGGNet16[17], AlexNet[18] 총 4개의 DNN을 사용하여 결과를 확인한다. DenseNet201, ResNet152, VGGNet16, AlexNet은 각각 Dense, Res, VGG, Alex로 나타내며 괄호안의 숫자는 그 DNN의 개수를 의미한다.

2. 동일 종류의 멀티 DNN 실행

이 실험에서는 동일 종류의 DNN 모델을 동시에 실행했을 때의 각 환경에서의 실행시간을 비교한다. 그림 9는 각각 15개의 네트워크를 실행했을 때 총 실행시간을 비교하고 있다. 예를 들면 x축의 Alex(15)는 Alex 모델 15개가 동시에 실행된 후 맨 마지막 모델이 종료되는 시점을 의미한다. 그림 9의 Res, VGG, Alex의 결과를 Q-S기준으로 보면 Res(15)의 실행시간은 M-C대비 약 21% 감소하였고 , S-C대비 4.8% 감소하였다. VGG(15)의 실행시간은 M-C대비 40.8%, S-C 대비 22.8% 감소하였고 Alex(15)의 실행시간은 M-C 대비 26.5%, S-C 대비 22.6% 감소한 것을 확인할 수 있다. 이 세 개의 실험군을 보면 M-C의 컨텍스트 교환 오버헤드가 가장 크게 영향을 주는 것을 알 수 있다.

Dense(15)의 실행시간을 보면 Q-S는 M-C 대비 약 14.7% 감소하였고 S-C대비 45%나 감소하였다. 위의 세 실험군과 다르게 M-C보다 S-C의 실행시간이 더 길게 나타났다. 이는 Dense(15)에서는 컨텍스트 교환 오버헤드보다 싱글 컨텍스트에서 mutex로 인한 오버헤드의 영향이 더 크다고 볼 수 있다.

오버헤드의 영향을 확인하기 위해 그림 10을 이용한

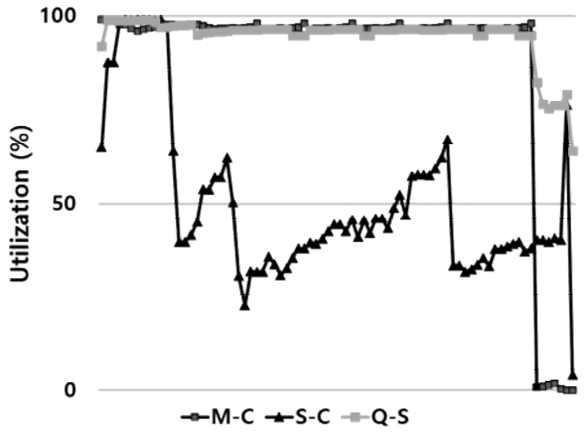

그림 10. 시간에 따른 Dense(15)의 GPU 사용율
Fig. 10. GPU utilization of Dense(15) over time.

다. 그림 10의 x축은 실행시간을 0%에서 100%로 나타낸 것이고 y축은 각 구간에서 GPU 사용률을 나타내며 Dense(15)를 대상으로 그래프로 나타냈다. 그래프를 보면 M-C와 Q-S는 실행하는 내내 거의 100%에 가까운 GPU 사용률을 보인다. 그림 10에는 없지만 나머지 Res, VGG, Alex 의 각 환경에서도 GPU를 거의 100% 사용하고 있다.

하지만 Dense(15)의 S-C는 약 20%실행한 구간부터 GPU사용량이 50%로 현저히 떨어진다. Dense의 특성상 레이어의 수는 다른 DNN에 비해 가장 많기 때문에 mutex를 거는 횟수가 더욱 많아진다. 추가로 각 레이어당 연산량도 작기 때문에 더 빠르게 커널함수를 발행해 주어야 하지만 이는 mutex로 인해 제한되므로 GPU사용량이 적은 것을 알 수 있다.

3. 서로 다른 종류의 멀티 DNN 실행

이 실험에서는 다른 종류의 DNN 모델을 동시에 실행했을 때 각 환경에서의 실행시간을 비교한다. 실행시간이 제일 짧은 Alex를 제외하고 나머지 세 개의 DNN을 조합하여 실행한다. 그림 11은 본 실험의 결과를 나타내며 x축의 숫자는 순서대로 (Dense, Res, VGG)의 갯수를 나타낸다.

멀티 DNN의 각 조합의 실행시간 결과를 보면 (8, 0, 8)은 M-C와 S-C의 결과가 거의 같고 Q-S와 비교했을 때 Q-S의 실행시간이 약 27.8% 감소하였다. (5, 5, 5)에서는 Q-S는 나머지와 실행시간을 비교했을 때 M-C대비 15%, S-C대비 18.5%감소하였고, (8, 8, 0)은 M-C대비 22%, S-C대비 16% 감소하였다.

실험 결과를 정리하면 동일 종류 멀티 DNN 실행과

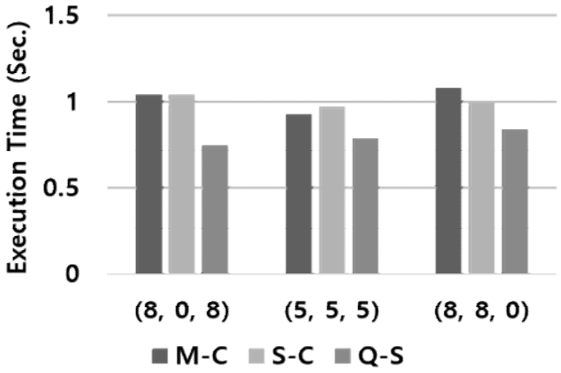

그림 11. 실행 환경에 따른 다른 종류 DNN의 실행 시간
Fig. 11. Execution times for different types of DNNs depending on the execution environment.

다른 종류 멀티 DNN실행에서 M-C와 S-C의 실행시간을 비교했을 때 항상 M-C의 실행시간이 크게 나오는 것이 아니라 S-C의 실행시간이 큰 경우도 있었다. 따라서 동시에 실행되는 DNN의 종류와 조합에 따라 컨텍스트 스위칭 오버헤드와 mutex 오버헤드의 영향을 다르게 받는 것을 알 수 있었다. 그리고 본 논문에서 제안한 Q-S와 M-C, S-C를 비교했을 때 동일 종류 멀티 DNN실행과 다른 종류 멀티 DNN 실행 모두 Q-S가 M-C, S-C에 비해 전체적으로 실행시간이 줄어드는 것을 확인할 수 있었다.

Ⅵ. 결 론

본 논문에서는 GPU 실행 환경을 멀티 컨텍스트 환경, 싱글 컨텍스트 환경으로 나누고 멀티 DNN이 실행될 때 멀티 컨텍스트 환경의 컨텍스트 교환 오버헤드와 싱글 컨텍스트 환경의 쓰레드 간 mutex 오버헤드를 분석하였다. 분석된 결과를 바탕으로 싱글 컨텍스트 환경에서 쓰레드 간 mutex로 인하여 GPU에서 레이어 연산이 지연됨을 막을 수 있는 프레임워크를 제안하였다. 이와 더불어 임베디드 GPU 기반 시스템에 기존 메모리 복사 방식을 사용하면 하나의 시스템 메모리에 같은 데이터가 두개 존재하게 되므로 CUDA의 제로카피 기술을 이용하여 하나의 메모리 공간을 공유할 수 있도록 하였다. 제로카피로 인한 GPU의 캐시미스는 타겟 디바이스의 하드웨어적 캐시 동기화 기법으로 피할 수 있음을 확인했다. 제안된 프레임워크를 상용 보드인 Xavier에 적용하고 실험한 결과 멀티 컨텍스트 환경 대비 실행시간이 최대 40.8%만큼 감소하였고, 싱글 컨텍스트 환경 대비 최대 45%감소하였다.

REFERENCES

[1] M. Vandersteegen, K Vanbeeck, and T goedeme, “Super accurate low latency object detection on a surveillance UAV”, in Proceedings of the 16th International Conference on Machine Vision Applications (MVA), Tokyo, Japan, May 2019.

[2] J. Barthélemy, N. Verstaevel, H. Forehead, and P. Perez, “Edge-Computing Video Analytics for Real-Time Traffic Monitoring in a Smart City“, MDPI, May 2019.

[3] https://developer.nvidia.com/drive

[4] https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit

[5] https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#multiple-contexts

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

[7] https://docs.nvidia.com/cuda/cuda-driver-api/index.html

[8] https://docs.nvidia.com/cuda/cuda-runtime-api/index.html

[9] https://developer.nvidia.com/cudnn

[10] https://docs.nvidia.com/cuda/cublas/index.html

[11] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#streams

[12] T. Amert, N. Otterness, M. Yang, J. H. Anderson and F. D. Smith, "GPU scheduling on the NVIDIA TX2: Hidden details revealed", Proc. IEEE Real-Time Syst. Symp. (RTSS), pp. 104-115, Dec. 2017.

[13] https://www.fastcompression.com/blog/jetson-zero-copy.htm

[14] https://developer.ridgerun.com/wiki/index.php?title=NVIDIA_CUDA_Memory_Management

[15] X. Yu, N. Zeng, S. Liu and Y.Zhang, “Utilization of DenseNet201 for diagnosis of breast abnormality”, in Machine Vision and Applications. vol. 30, Oct. 2019.

[16] L. Nguyen, D. Lin, Z. Lin and J. Cao, “Deep CNNs for microscopic image classification by exploiting transfer learning and feature concatenation”, in Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS), Florence, Italy, 2018.

[17] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” in Proceedings of the International Conference on Learning Representations, San Diego, CA, 2015.

[18] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” in Proceedings of the 26th Conference on Neural Information Processing Systems, Lake Tahoe, pp. 1097 - 1105, 2012.

저 자 소 개

임 철 순(학생회원)

2021년 한성대학교 IT응용시스템 공학과 학사 졸업.

2021년～현재 한성대학교 IT융합 공학과 석사과정.

<주관심분야 : GPU Programing, DNN framework, Deep Learning>

김 명 선(정회원)

2000년～2002년 LG전자 액세스네트워크 연구소 주임연구원

2002년～2011년 삼성전자 DMC 연구소 책임연구원

2016년 서울대학교 전기컴퓨터 공학부 박사 졸업.

2016년～2019년 삼성전자 SR연구소 수석연구원

2019～현재 한성대학교 IT융합공학부 조교수

<주관심분야 : NPU(인공지능 가속기), Linux kernel, HW/SW Co-desgin 등>