컨테이너 상 GPU 성능 분석과 멀티태스크 스케줄링 특성 분석

손민지[O1] 김현준[2] 한환수[1]
[1]성균관대학교 소프트웨어대학
[2]성균관대학교 정보통신대학
{sonmj426, hjunkim, hhan}@skku.edu

Analyzing GPU Performance on Containers and Characteristics of Multi-Task Scheduling

Minji Son[O1] Hyunjun Kim[2] Hwansoo Han[1]
[1]College of Software, Sungkyunkwan University
[2]College of Information and Communication Engineering, Sungkyunkwan University

요 약

Docker는 컨테이너 기반의 오픈소스 가상화 플랫폼으로서 다양한 프로그램과 실행환경을 컨테이너로 추상화하고 동일한 인터페이스를 제공하여 프로그램의 배포 및 관리의 단순화를 가능하게 한다. 하나의 머신에서 여러 가상화 환경을 제공하는 기술의 수요가 증가함에 따라 운영체제 전체를 가상화해야 하는 가상머신에 비하여 운영체제의 일부만을 가상화하는 컨테이너는 가볍다는 장점이 있어 널리 쓰이고 있다. 최근 딥 러닝의 중요성이 높아지면서 GPU의 중요성 또한 갈수록 높아지고 있다. 따라서 본 논문에서는 시간 및 메모리 측면에서 NVIDIA Docker의 성능을 분석하여 이를 바탕으로 성능을 개선할 방안을 탐색한다.

1. 서 론

소프트웨어는 다양한 환경들이 존재하고 요구되기 때문에 그만큼 다양한 변수를 고려하여 개발되어야 한다. Docker는 컨테이너라는 격리된 공간에서 프로세스를 동작함으로써 뛰어난 이식성을 제공하며 환경에 구애받지 않고 애플리케이션을 신속하게 배포하고 서버를 효율적으로 운영할 수 있도록 한다. 기존의 컨테이너를 활용한 기술들에 비해 Docker는 다음과 같은 장점들을 지닌다. 우선 사용법이 간편하며 Docker hub를 통한 다양한 이미지 검색 및 사용이 용이하다. 적은 자원을 효율적으로 사용하여 가볍고 빠르며 애플리케이션의 기능들을 각각의 컨테이너로 쉽게 분리할 수 있기 때문에 뛰어난 모듈성과 확장성을 지닌다.

본래 빠른 그래픽 처리를 위해 설계된 보조-프로세서 GPU는 강력한 병렬 연산 능력을 갖추고 있어 최근 딥 러닝 분야의 부상으로 그 중요성이 강조되고 있다. 일련의 명령어를 순차적으로 하나씩 처리하는 CPU에 비해 GPU는 여러 명령어들을 동시에 처리하는 병렬 처리 방식에 특화된 구조를 지닌다. 따라서 수많은 연산을 한꺼번에 수행할 수 있는 GPU가 인공지능, 자율주행, 빅데이터 등의 분야에서 주목받고 있다.

본 논문에서는 Docker 컨테이너 내에서 NVIDIA GPU를 이용할 수 있도록 해주는 NVIDIA Docker의 성능을 시간 및 메모리 측면에서 분석한다. 결과 분석을 통해 NVIDIA Docker의 성능 개선안으로 통합 메모리(unified memory)와 연산 선점(compute preemption) 기능이 제공되는 최신 GPU 사용을 제안한다. 이를 통해 메모리 접근의 오버헤드를 줄이고 폭넓은 메모리 사용이 가능하기를 기대한다.

본 논문은 한국 연구재단의 지원을 받아 수행된 연구결과임 (NRF-2016M3C4A7952587)

이후 내용은 다음과 같이 구성된다. 2장에서 본 논문의 이해를 위해 Docker, NVIDIA Docker, CUDA 실행 모델 및 메모리 모델, MPS에 대한 배경 지식을 제공하고 3장에서는 실험 환경, 설계 및 결과에 대하여 설명한다. 마지막으로 4장에서 본 논문의 결론을 내린다.

2. 배경 지식

2.1. Docker

Docker [1]는 namespaces, control groups와 같은 리눅스 커널 기능을 이용한 리눅스 컨테이너를 기반으로 시작한 오픈소스 가상화 플랫폼이다. 컨테이너라는 격리된 공간을 제공하고 이들이 서로 충돌하지 않도록 하기 위해 namespaces 기능을 사용하며 이는 pid, mnt, net 등의 네임스페이스를 지원한다. Control groups는 CPU, 메모리, 네트워크 등의 자원에 대한 제어를 가능하게 해준다. 즉, 각각의 컨테이너가 필요한 자원만을 사용하도록 자원을 제어하고 제한한다.

컨테이너는 code, runtime, system tools, system libraries 등 어떠한 애플리케이션이 실행 환경과 관계없이 동일하게 실행되는데 필요한 모든 요소를 포함한다. 가상 머신과는 다르게 컨테이너는 유저 공간을 추상화하고 호스트 시스템의 커널을 다른 컨테이너들과 공유하기 때문에 매우 가볍고 빠르며 이식성이 높다는 장점이 있다.

2.2. NVIDIA Docker

Docker 컨테이너는 어떠한 운영 체제나 프로세서의 조합인지에 대한 아무런 지식이 없더라도 상관없이 기능을 수행할 수 있음(platform-agnostic)은 물론 어떠한 하드웨어에도 종속적이지 않다

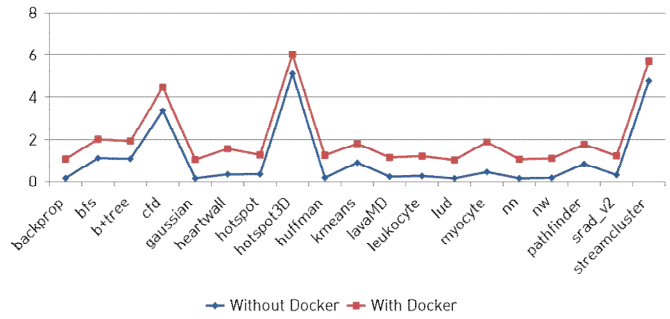

그림 1. Docker 사용 여부에 따른 실행 시간 차이

(hardware-agnostic). 따라서 Docker는 NVIDIA GPU와 같이 특수한 하드웨어를 기본적으로 지원하지 않는다.

NVIDIA Docker [2]는 NVIDIA GPU를 활용하는 Docker 이미지 이식을 가능하게 하기 위해 개발되었으며 GPU 기반 컨테이너 이식을 위해 필요한 중요한 요소 다음의 2가지를 제공한다. 첫 번째는 드라이버 종류에 상관없이 실행되는 CUDA 이미지이다. CUDA [3]는 GPU를 그래픽처리가 아닌 범용 목적의 연산에 사용하기 위한 프로그래밍 모델과 API를 제공하는 병렬 연산 플랫폼이다. 두 번째는 Docker 상위에서 일부 명령들을 포착하여 기존의 Docker 명령이 아닌 nvidia-docker 명령이 실행되도록 옵션을 추가해주는 래퍼(wrapper)이다.

2.3. GPU CUDA 실행모델 및 메모리 모델

GPU는 긴 메모리 지연시간을 숨기기 위해 수천에서 수만개의 쓰레드를 동시에 동일한 명령어를 수행하며, 이를 SIMT (Single Instruction Multiple Thread) 모델이라고 한다. CUDA에서는 일반적으로 32개의 쓰레드 그룹을 워프(warp)라고 부르고, 다수개의 워프 그룹을 쓰레드 블록(thread block)이라고 한다. 각 쓰레드 블록은 GPU 내부의 멀티프로세서에 할당되며, 각 멀티프로세서 내부에서는 할당 받은 쓰레드 블록을 다시 워프 단위로 구분하여 명령어를 수행한다.

GPU는 온/오프-칩 메모리로 구분된다. 온-칩 메모리에는 레지스터 파일, 다중-레벨 캐시, 그리고 공유 메모리가 있다. 오프-칩 메모리에는 전역 메모리, 상수 메모리, 지역 메모리, 그리고 텍스처 메모리가 있다. 상수 메모리와 텍스처 메모리는 읽기 전용이며, 전역 메모리에 대해서는 읽기/쓰기가 가능하다. 지역 메모리는 레지스터 스필(register spill)이 발생하였을 때 사용하는 메모리 공간이다. 오프-칩 메모리는 일반적으로 온-칩 메모리 대비 최대 100배 정도의 긴 접근 시간을 필요로 한다. CUDA 프로그래밍에서는 기본적으로 커널 실행 전에 CPU 메모리의 데이터를 GPU의 전역 메모리로 복사하고 사용한다. 통합 메모리(unified memory) 기능을 이용하면 커널 수행 중에 필요한 데이터를 CPU 메모리로부터 복사하여 사용하는 기능도 제공하고 있다.

2.4. Multi-Process Service (MPS)

MPS는 여러 프로세스가 단일 GPU에서 자원을 동시에 공유할 수 있도록 하는 서비스이다. GPU의 스케줄러는 GPU에서 한 번에 하나의 CUDA 컨텍스트만 시작할 수 있는데 이는 GPU 자원을 포화시키기에 충분하지 않은 작업을 수행할 경우에 GPU를 충분히

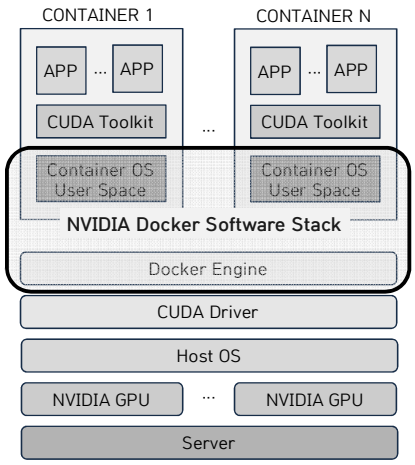

그림 2. NVIDIA Docker 구조

활용하지 못하는 원인이 된다. MPS 서버 프로세스는 여러 CUDA 컨텍스트들의 작업을 하나의 CUDA 컨텍스트로 모아주는 깔때기 역할을 수행한다. 이를 통해 MPS는 다른 프로세스들의 커널과 메모리 복사 작업이 동시에 수행될 수 있도록 하여 높은 활용률과 짧은 실행 시간을 제공한다.

3. 실험 설계 및 결과 분석

본 장에서는 실험이 진행된 환경과 사용된 벤치마크에 대하여 기술한다. 이어서 3.2에서는 Docker를 사용하였을 경우 시간적 측면에서 발생하는 오버헤드를 측정하고, 3.3에서는 Docker에서 하나의 GPU를 대상으로 여러 컨테이너를 동시에 수행하도록 하였을 때 발생할 수 있는 공유 자원(전역 메모리) 경합에 대한 추세를 확인하는 실험을 진행하였다. 3.4에서는 MPS의 사용 여부에 따른 멀티태스크 스케줄링 특성을 분석하였다.

3.1. 실험 환경 및 벤치마크

본 실험이 진행된 시스템은 NVIDIA GTX980Ti (Maxwell 아키텍처) GPU와 V100 (Volta 아키텍처)으로 구성되어 있으며 Ubuntu 16.04.1에서 수행하였다. GTX980Ti GPU는 드라이버 버전 387.26과 CUDA 버전 8.0.61을 사용하였고, 총 6GB가 탑재되어 있다.

본 실험은 Rodinia_3.1와 Polybench/GPU 1.0 벤치마크 애플리케이션을 대상으로 NVIDIA Docker version 2.0.2 성능을 측정하였다. Rodinia는 GPU 성능 측정을 위해 많이 쓰이는 실제 애플리케이션들로 구성된 벤치마크로서 병렬 통신 형태, 동기화 기법과 동력 소비량에 대하여 광범위하게 다룬다[5]. Polybench는 벡터 및 행렬 곱셈, 공분산(covariance) 등 다면체 관련 벤치마크들로 구성되어 있다.

3.2. NVIDIA Docker의 소프트웨어 스택 오버헤드 분석

Rodinia 벤치마크 애플리케이션 19개를 대상으로 하여 Docker 사용 여부에 따른 실행 시간 차이를 분석해본 결과 그림 1과 같은 양상을 보인다. Docker를 사용한 경우에는 외부에서 컨테이너 안의 명령을 실행하는 docker exec을 사용하여 애플리케이션을 실행하였으며 실험 결과 값은 각 애플리케이션을 Docker를 사용

표 1. MPS 사용 여부에 따른 실행 시간 차이 (단위: 초)
(CS: Cache sensitive 커널, CI: Cache insensitive 커널)

	CS		CI	
입력 크기	MPS 사용	MPS 미사용	MPS 사용	MPS 미사용
5,120	0.73984	1.4952	0.70990	1.49199
10,240	2.72925	3.24161	1.58436	3.35677
20,480	13.38374	12.57792	6.75526	12.37545
40,960	223.60555	61.51608	40.67550	50.70506

할 경우와 사용하지 않을 경우 각각 5회 실행하고 그 평균치를 내었다. 그 결과, Docker를 사용할 경우에 Docker를 사용하지 않은 경우보다 실행 시간이 평균 0.96초 정도의 증가를 보였다. 이러한 차이는 애플리케이션의 종류나 실행 시간에 영향을 받는 것이 아니라 그림 2에 나타난 NVIDIA Docker 소프트웨어 스택의 추가로 인해 발생하는 오버헤드이다. 비록 Docker를 사용할 경우 실행 시간이 다소 증가하지만, 이는 Docker가 제공하는 이점들을 감안한다면 충분히 감수할 수 있는 오버헤드이다.

3.3. GPU의 제한된 디바이스 메모리 크기에 따른 한계

본 실험은 Docker 컨테이너 개수를 점차 늘려가며 각 컨테이너에서 커널 시간이 비교적 긴 Rodinia 애플리케이션 중 임의로 선택한 hotspot3D를 실행하는 방식으로 진행하였다. 하나의 hotspot3D 애플리케이션은 98MB만큼의 메모리를 사용한다. 해당 애플리케이션의 경우, 52개의 컨테이너를 동시 운용하여 총 GPU 메모리의 약 93%를 사용할 때까지는 모든 애플리케이션이 원활하게 실행되었으나, 이를 초과할 시 오류가 나고 대부분의 애플리케이션이 실행에 실패하였다.

메모리의 경우, Pascal GPU 아키텍처부터 제공되는 특징인 통합 메모리(unified memory)[6]를 활용한다면 성능을 크게 개선할 수 있기를 기대한다. 통합 메모리는 CPU와 GPU 메모리를 위해 하나로 매끄럽게 통합된 가상 주소 공간을 제공함으로써 GPU와 CPU 전체 가상 주소 공간 사이에서의 데이터 이동이 투명하게 이루어질 수 있도록 해준다. 더불어 Pascal GPU 아키텍처부터는 통합 메모리 공간이 가상메모리 주소로 관리되고 있으며, GPU 디바이스 메모리보다 큰 공간을 할당하고 사용하는 것이 가능하다. 따라서 Docker에서 전역 메모리 공간에 할당되어 실행이 불가능한 커널들을 대상으로 통합 메모리를 사용하도록 변환함으로써 메모리 공간 부족으로 인한 프로그램 실패와 같은 에러를 피할 수 있다.

3.4. 멀티태스크 스케줄링 특성 분석

본 실험은 하나의 애플리케이션을 서로 다른 특성을 갖는 두 애플리케이션과 함께 실행한 결과를 두 환경에서 분석한다. L1 cache 크기에 따른 cache hit rate를 비교해서 cache의 크기가 32KB에서 64KB으로 증가했을 때 cache hit rate가 10% 이상 증가했다면 해당 애플리케이션의 커널이 cache sensitive(CS)하다고 판단하였다. 나머지 애플리케이션들에 대해서는 cache insensitive(CI)하다고 분류하였다. 표 1은 CS한 애플리케이션을 각각 CS한 애플리케이션, CI한 애플리케이션과 함께 실행할 때 MPS 사용 여부에 따른 실행 시간 차이를 보여준다. CI와 함께 실행한 경우에는 두 애플리케이션 모두 GPU 자원(예. cache, CUDA core 등)을 포화시킬 만큼의 작업을 수행하지 않기 때문에 MPS를 사용할 때가 항상 더 짧은 실행 시간을 보인다. 하지만 CS와 함께 실행한 경우에는 GPU가 입력 크기가 10,240일 때부터 cache가 포화 상태에 이르러 입력 크기가 20,480일 때부터는 MPS를 사용하지 않을 때가 더 짧은 실행 시간을 보이게 된다.

GPU 자원에 여유가 있을 때의 MPS 사용은 GPU의 활용률 증가와 짧은 실행 시간을 제공하지만 GPU 자원이 포화 상태에 이르면 급격한 성능 저하를 보인다. 따라서 GPU 자원 사용률을 사전에 분석할 수 있는 도구가 필요하며 그 결과를 통해 MPS의 사용 여부를 결정한다면 여러 프로세스를 실행할 때 GPU를 보다 효율적으로 공유할 수 있을 것이다.

4. 결 론

신속한 배포와 효율적인 서버 운영을 위해서는 Docker가 필요하며 딥 러닝의 부상으로 GPU의 중요성이 부각되고 있다. 본 논문에서는 Docker와 GPU를 모두 활용하는 NVIDIA Docker에 대한 성능 분석을 수행하였다. 총 세 가지의 실험을 통해 1) NVIDIA Docker의 소프트웨어 스택 오버헤드 수준을 확인하였고, 2) 하나의 GPU 디바이스를 대상으로 다중 컨테이너가 공유하여 수행될 때 발생하는 GPU 디바이스 메모리 오류를 분석 및 확인하였으며, 3) MPS의 사용 여부에 따른 멀티태스크 스케줄링 특성을 분석하였다.

Pascal GPU 아키텍처는 기존 Maxwell과 Kepler GPU 아키텍처에서 제공하는 쓰레드 블록 단위에서의 연산 선점 대신 명령어 단위에서의 연산 선점(compute preemption)이 가능하다. 이러한 연산 선점은 장시간 실행되는 애플리케이션이 시스템을 독점하거나 제한 시간을 초과하는 상황을 방지하며 효율적인 스케줄링을 가능하게 한다. 향후 이러한 이점을 활용하여 여러 컨테이너에서 여러 GPU를 공유하는 환경에서 발생할 수 있는 공유 자원 경합 문제를 분석하고 개선할 계획이다.

참고문헌

[1] D. Merkel, “Docker: lightweight linux containers for consistent development and deployment,” *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[2] “GPU-Enabled Docker Container.” http://www.nvidia.com/object/docker-container.html, accessed: 2018-05-08.

[3] “CUDA Toolkit,” https://developer.nvidia.com/cuda-toolkit, accessed: 2018-05-08.

[4] https://github.com/NVIDIA/nvidia-docker, Official GitHub repository of NVIDIA Docker. Accessed: 2018-05-08.

[5] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H., Skadron, K., Rodinia: A benchmark suite for heterogeneous computing. In Proceedings of the IEEE International Symposium on Workload Characterization, 2009.

[6] Nvidia Tesla P100 white paper, 2017.