Abstract

여러 개의 컨테이너가 GPU를 효율적으로 나누어 사용할 수 있도록 ConVGPU를 제안

ConVGPU는 컨테이너 실행 시 요구되는 GPU 메모리를 보장하여 program failure 또는 deadlock 상황을 방지

이는 4가지 scheduling algorithm을 통하여 실현

1. Introduction

다양한 유저들에게 서비스를 제공하는 cloud computing system에서 하나의 컴퓨터를 가상화된 환경으로 가상화하는 VM이 중요하게 여김

하지만 container-based virtualization이 가볍고 사용하기 용이하여 VM을 대신하기 시작함

OS 전체를 가상화시켜야하는 VM에 비하여 container는 OS의 일부만을 가상화시킴

빠른 build time, execution time, processing speed(가볍기 때문에)

또한 가볍기 때문에 한 machine은 적은 자원으로 더 많은 수의 가상환경을 제공할 수 있음

본래 빠른 graphics rendering을 위해 고안된 GPU는 각 pixel에 같은 알고리즘을 적용하는 parallel computing을 위한 특수한 구조를 갖고 있음

최근에는 이처럼 강력한 parallel computing 능력을 다양한 산업 그리고 연구 분야의 general-purpose high-performance computing에 사용(machine learning이나 성능 향상을 위해 cloud환경에서 사용됨)

GPU의 general-purpose use를 위한 플랫폼: CUDA & OpenCL

CUDA는 NVIDIA GPU를 위한 프로그래밍 모델과 API를 제공하는 parallel computing 플랫폼

가상화 시스템은 host의 hardware computing 자원들을 관리하고 제공

예를 들면, Docker는 cgroups & UnionFS를 사용

cgroups: 리눅스 커널이 제공. CPU 시간, 메모리, 네트워크 bandwidth 조절

Union FIle System: 디스크 storage 관리

가상화 환경에서 virtualIzed GPU를 제공하려는 많은 시도들이 있었음(CUDA API를 복사하는 방식으로)

하지만 GPU를 가상화시키는 과정에서 GPU의 성능을 저하시킨다는 단점이 모두 있음

또한 하드웨어 레벨에서 GPU를 가상화시키려는 시도도 있었지만 특정한 device가 필요하고 container-based virtualized 환경을 지원하지 않음

NVIDIA Docker는 전혀 다른 방식으로 접근함. GPU 전체를 하나의 컨테이너에게 할당함. 위에서 발생한 문제점들은 거의 해결하지만 같은 GPU를 여러 다른 컨테이너들이 동시에 사용하려고 할 때 치명적인 문제 발생.

container간의 효율적인 자원 공유를 위한 다양한 시도가 있었지만 그 중 GPU 자원을 공유하려는 연구는 없음. 그리하여 본 논문에서 ConVGPU 제안!

ConVGPU는 GPU자원의 공평한 사용을 위해 컨테이너 schedule.

실험을 통해 전체 실행 시간의 관점에서 best fit 알고리즘이 가장 빠르다는 것을 발견

1. Backgrounds
2. GPU & CUDA

parallel execution에 특화된 GPU의 특징 때문에 요즘은 general-purpose computing에 사용됨.

GPU에서 사용된 데이터들은 CPU로 복사되어야 하기 때문에(GPU는 부속적인 processing unit이기 때문에 CPU가 작업을 할당하는 등의 일들을 수행하기 위해 GPU에서 사용된 데이터들의 복사가 필요하다. 여기서 많은 성능 저하?가 발생한다.) GPU를 좀 더 쉽게 사용할 수 있도록 하는 플랫폼들이 나타남.

GPGPU를 위한 플랫폼 중 가장 널리 쓰이는 CUDA

두 가지 programming interface

runtime API: high-level. low-level driver API위에서 실행됨. driver API보다 implicit initialization and management를 제공하기 때문에 사용하기 쉬움.

하지만 driver API는 fine-grained context control and module loading을 수행할 수 있음.

kernel! GPU를 사용하는 user program에서 GPU에서 실행되도록 적힌 부분

CUDA를 사용한 모든 프로그램은 CUDA 플랫폼의 일부분이며 컴파일러인 nvcc로 컴파일되어야함.

1. GPU Virtualization

가상화 GPU를 제공하기 위해서 많은 framework들은 Remote-API기술을 사용함.

이는 가상화 환경 내에서의 API call을 포착한 뒤 이를 host system에게 redirecting해줌.

따라서 가상 환경과 호스트 사이의 소통 방식이 중요

중요 단점은 API 포착의 어려움. 또한 API에 변경 발생 시에도 추가적인 노력이 요구됨.

PCI pass-through technique는 GPU를 가상 환경에 바로 할당함.

VM에서는 CPU로부터의 IOMMU support가 필요함.

container based에서는 container로 device mount를 link함으로써 쉽게 구현됨.

OpenStack: Nova config file에 GPU의 device ID를 기록

NVIDIA Docker: --device option

장점: no performance degradation

하지만 연결된 GPU는 공유되지 못함

NVIDIA GRID: 하드웨어 레벨에서 GPU를 가상화

특정한 타입의 GPU가 필요하며 container based virtualization을 못함.

1. Docker

open source application container engine

공개 후 가장 널리 쓰이며 현재는 de-facto choice(법적으로 정해져 있지는 않지만 표준)

process sandboxing(security mechanism for separating running programs)를 위해 libcontainer를 사용

사실 docker는 기존에 있는 다양한 kernel기술들을 기반으로 만들어짐

cgroups를 사용하여 각 컨테이너에 속하는 프로세스들을 분리하고 그들의 CPU시간이나 메모리 제한을 관리

linux namespaces를 사용하여 개별적인 pid, mount point(mnt), hostname, domain name 갖기 위해?

1. NVIDIA Docker

docker container에서 NVIDIA GPU를 쉽게 사용하기 위한 utility set

두 가지 executable programs: nvidia-docker & nvidia-docker-plugin

nvidia-docker: docker의 꼭대기에 있는 thin wrapper로서 기존 docker 명령이 아닌 nvidia-docker command를 사용하도록 유저로부터의 명령을 포착

유저 명령을 해석하고 몇 가지 옵션을 추가하는 형식으로 수정하는 역할을 함

오직 run, create 명령만 포착

label 정보를 읽어 어떠한 이미지가 CUDA API를 사용하는지, GPU device 개수, GPU driver 버전 등을 확인함.

본 논문에서 GPU 메모리를 관리하는 기능을 추가하기 위해 nvidia-docker 수정함.

nvidia-docker-plugin: docker 기능 확장

NVIDIA GPU와 CUDA API의 존재 확인하고 적절한 버전의 binaries, library files 제공

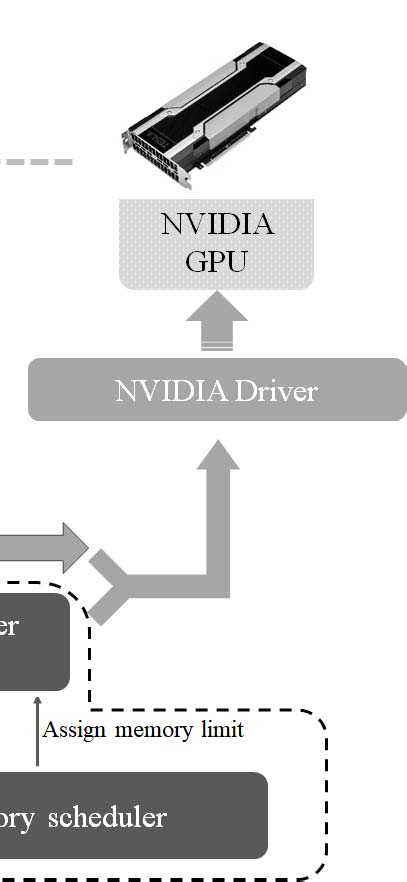
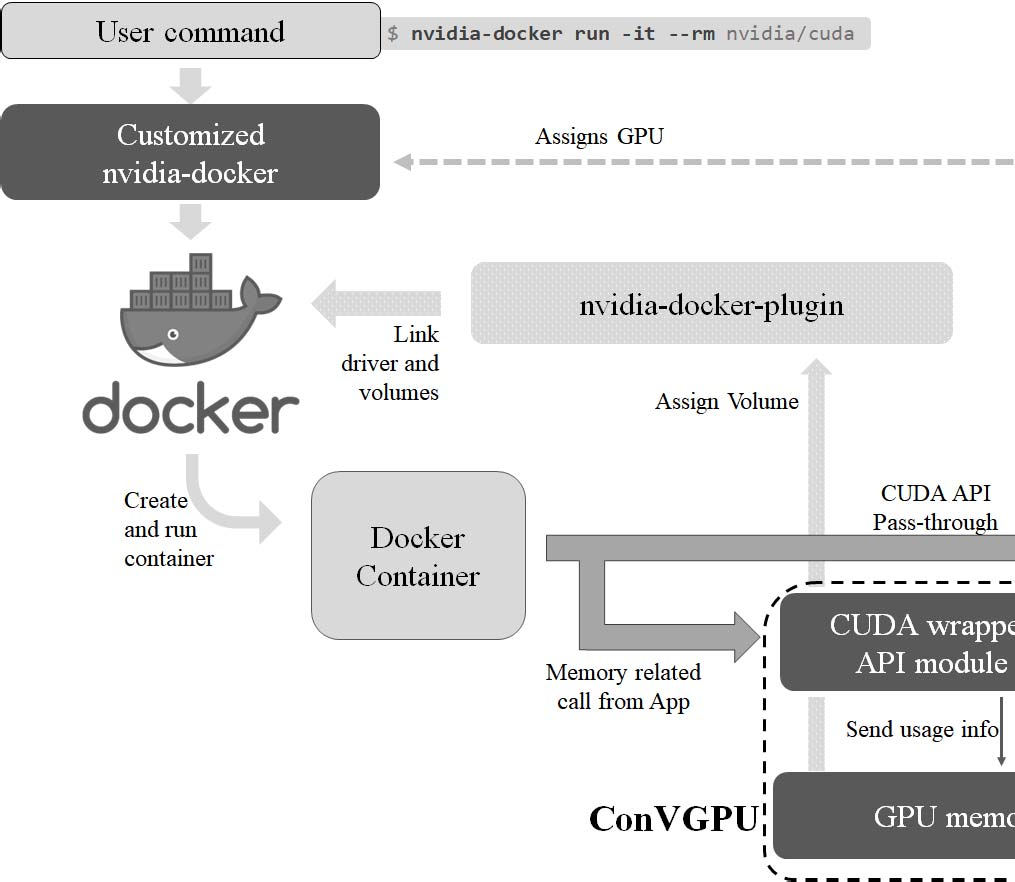
1. Design and Implementation

ConVGPU = GPU memory scheduler + CUDA wrapper API module

GPU memory scheduler는 각 컨테이너의 GPU 메모리 제한을 확인

CUDA Wrapper API module은 컨테이너 생성 시 삽입되어 그 안에서 유저 프로그램으로부터의 original CUDA API call들을 포착

1. System Design



ConVGPU에서 고려해야할 사항들

Isolation: 각 유저 프로그램이 GPU를 사용할 때 완전히 고립되어있음을 보장

Consistency: GPU 사용 중 충돌이나 기능 불량이 발생하지 않도록 한 컨테이너의 실패가 다른 컨테이너들에 영향을 주지 않도록 함. 또한 GPU메모리 용량보다 많은 컨테이너들이 생성되었을 경우에도 잘 동작하여야 함

Shareability: 최대한 기존 CUDA API 사용(공개되지 않은 특성들도 있고 compatibility를 위해)

host와 containerized 환경 사이에서의 소통은 UNIX socket으로

third party가 정보를 가로챌 수 있는 shared memory & plain file share

높은 complexity 와 낮은 performance를 보이는 TCP/IP socket

1. Customized NVIDIA Docker

컨테이너가 쓸 수 있는 최대 GPU 메모리 설정

--nvidia-memory=<size> custom option

옵션이 없다면 도커 이미지 내 label인 com.nvidia.memory.limit:<size>

label도 없다면 1 GiB

이러한 정보는 컨테이너가 만들어지기 전에 scheduler에 전달됨

--volume option으로 container와 directory(CUDA wrapper API module과 UNIX socket이 있는) 연결

--env option으로 CUDA wrapper API module이 CUDA API module보다 먼저 load될 수 있도록 환경 변수 설정

컨테이너의 stop 상태(Exit?)를 감지하기 위해 컨테이너에 nvidia-docker-plugin과 연결된 dummy volume을 추가하도록 함

컨테이너 종료 시 도커는 volume을 unmount하고 이를 통해 nvidia-docker-plugin은 컨테이너가 종료되었다는 것을 알고 scheduler에게 close 신호를 보냄

1. CUDA wrapper API module

“API capturing module”

컨테이너의 관리 및 스케쥴링, GPU 메모리 사용 조절을 위한 Linux shared library

보통 memory allocation/deallocation API 관련

해당 shared library로의 path를 LD\_PRELOAD 환경변수에 추가하면 override original CUDA API & intercept API calls

여기 너무 어려우ㅓ....

차별점! CUDA API 전체를 복사하지 않음

일부만 override, 이식성?이 좋음, driver API & runtime API(원래는 둘 중 하나만인ㄱㅏ?)

<http://docs.nvidia.com/cuda/cuda-runtime-api/driver-vs-runtime-api.html>

포착되는 API 종류는 크게 2가지로 나뉨 (allocation & deallocation)

Allocation API: 할당하려고 할 때 wrapper module은 요구된 메모리 사이즈만 알고 있음

이에 대한 정보를 스케줄러에게 전달&확인 -> 가능하다면 original CUDA API로 메모리 할당한 후 여러 정보들을 스케줄러에게 다시 전달 (memory address, current pid, size info)

참고)NVIDIA GPU 메모리 종류 : <http://mkblog.co.kr/2016/11/26/nvidia-gpu-memory-types/>

일부 allocation API는 처음 요구된 메모리보다 큰 사이즈를 할당하기도 함 -> 이런 경우에는 wrapper module이 가능한 메모리를 확인하기 전에 변경된 사이즈를 파악함(계산을 한다는건지 정보를 받아오는건지는 잘 이해하지 못하겠다)

Deallocation API: wrapper module은 메모리 주소를 안다! original CUDA API로 deallocate하고 이를 스케줄러와 유저 프로그램에게 알림

프로그램 종료 시 \_\_cudUnregisterFatBinary CUDA API가 발동됨 (GPU로부터 CUDA fat binary를 unregister함) -> wrapper module은 해당 API를 포착하여 스케줄러에게 현재 process가 쓰는 memory를 deallocate함

1. GPU memory scheduler

Go 프로그래밍 언어로 작성됨

매 GPU memory allocation마다 허락할지 멈출지 거절할지 결정

각 컨테이너를 위한 UNIX socket을 만들어 컨테이너 내의 wrapper module과 소통

\*\*(실행 전 nvidia-docker가 요청, to share the volume with the container) unique directory -> 해당 directory 안에서 UNIX socket 만들기 -> wrapper module을 복사해옴\*\*

앞에서 언급된 wrapper module과의 소통 과정 설몀

실제 allocation이 발생하고 할당된 주소와 메모리 사이즈 정보가 전달되면 스케줄러는 이를 hash 구조로 관리하고 총 메모리 사용량을 계산함(각 과정은 mutex lock으로 race condition 방지)

race condition이란! 두 명령어가 동시에 같은 기억 장소에 접근하여 그들 사이의 경쟁에 의해 수행 결과를 예측할 수 없는 경우

64MiB (현재 process에 관련된 정보들) + 2MiB (CUDA context)

스케줄러는 해당 pid로부터 요청된 첫 allocation인지 확인한 후 추가적인 66MiB 할당

프로그램이 끝났다는 것을 wrapper module이 감지하고 이를 스케줄러에게 알리면 해당 pid로부터의 memory allocation 정보와 컨테이너로부터의 모든 정보를 지움

컨테이너가 적절한 크기의 GPU 메모리를 요구했지만 전체 시스템의 메모리 크기가 충분하지 않을 때는 기다려야함 -> 스케줄링이 필요함!

1. FIFO (First-in, first-out)
2. BF (Best-fit): 남은 메모리를 넘지않지만 부족한 정도가 가장 가까운 컨테이너, 없다면 가장 적은 크기가 부족한 컨테이너
3. RU (Recent use)
4. Rand (Random)
5. Experimental Evaluation
6. Experimental setup

GPU driver version 375.51 with CUDA version 8.0.44

NVIDIA Docker version 1.0.0 RC 3

docker version 1.12.3

SINGLE (container creation + API response + program running)

wrapper module과 연결한 거의 모든 API call (같은 기능을 수행하지만 형식만 조금 다른 API들을 일부 제외함)

**clock\_gettime POSIX.1 function with CLOCK\_MONOTONIC clock**

MULTIPLE

GPU 메모리 크기에 따라 분류하여 스케줄링 알고리즘 평가

We emulated the cloud

usage by choosing the type of the containers randomly and

running it every five seconds.???

각 컨테이너는 최대 GPU memory를 allocate하고 같은 크기의 CPU memory도 allocate함

SAMPLE PROGRAM: CPU로부터 dummy data를 GPU로 복사해와서 계산하고 다시 결과를 CPU로 return함 -> 크기에 따라 소요되는 시간이 5초부터 45초까지 달라짐

We changed the number of the

containers from 4 to 38 and measured the finished time of

all containers and suspended time of each container. All tests

are repeated 6 times and the average value is used.

1. Single container performance evaluation

API RESPONSE TIME

보통 2배씩 더 걸림 <- scheduler와 wrapper module 사이의 communication

특히 cudaMallocPitch는 첫 실행 때는 현재 GPU의 pitch size를 retrieve받아와야 하기 때문에 3배 넘게 차이남

cudaMallocManaged는 CPU와 GPU 메모리를 같은 메모리 주소로 할당해야해서? 40배 정도의 시간이 더 소요됨 <- mapped memory 사용 (longer response time?)

cudaFree는 GPU에서의 연산 시간이 다른 API들보다 적어 시간도 적게 소요됨

cudaMemGetInfo는 ConVGPU가 이미 해당 API의 return 값을 알고있기 때문에 0.01millisecond적게 걸림 (marks every memory information from the container)

CONTAINER CREATION TIME

스케줄러가 GPU 메모리를 확인하고 할당하는 시간 때문에 15%(0.0618sec)정도 오래 걸림

PROGRAM RUNNING

비록 response 시간이 2배 정도 더 걸리지만 이는 전체 runtime에 거의 영향을 주지 않음(0.7% 더 걸림)

유저 프로그램은 대부분의 시간을 copying data from/to the CPU memory and running GPU kernel code?? (그래서 api 수행 시간은 별로 중요하지 않다는건ㄱㅏ...

1. Scheduling algorithms performance evaluation

FINISHED TIME COMPARISON

컨테이너 크기를 임의로 고르는데도 불구하고 컨테이너 수가 2배되면 끝나는 시간도 대략 2배

컨테이너 수가 16보다 작을 때까지는 4가지 모두 비슷한 performance를 보이지만 18을 넘어가기 시작하면 best-fit algorithm이 다른 algorithm보다 평균적으로 30초 빨라짐 <- GPU memory throughput을 최대화하기 때문에

그리고 random이 제일 안좋은 성능을 보임

AVERAGE SUSPENDED TIME COMPARISON

24까지는 큰 차이를 안 보이다가 26부터 best-fit이 평균적으로 15초 더 걸림

starving may occur if there is no same size matched among the running containers.

따라서 best-fit이 전체overall task 수행할 때는 제일 빠르지만 각 컨테이너를 위한 waiting 시간이 평균적으로 더 필요하다는 것을 알 수 있음

V. CONCLUSIONS AND FUTURE STUDY

container based virtualized environment에서는 software관점에서 gpu를 가상화하는 방식이 맞지 않음 -> gpu전체를 가상화하려고 한 것이 실패의 주원인

ConVGPU: isolates the GPU memory, allows the container to use only a fraction of GPU memory.