

Summer Research Report

ZHANG HUAKANG

July 7, 2021

1 Introduction

1.1 Background

The concept of cloud computing was first proposed in 2005 [1] which means that the processing of big data is transferred to the server in the data center.. In 2006, Amazon's IaaS platform AWS was launched. Since then, cloud computing has profoundly changed human life. Internet of Things (IoT) was first proposed in 1999 [2]. With the development of IoT, large amounts of data are collected, uploaded and processed in the data center. In recent years, deep learning as a part of a broader family of machine learning methods based on artificial neural networks with representation learning is growing rapidly and is widely used in areas such as computer vision and natural language processing. Combined with IoT, it has given rise to many applications that have changed people's lives such as autonomous vehicle. But, due to the network bandwidth limitations and latency, centralized cloud computing architecture becomes inefficient and the computational task of deep learning is shifting from central servers to edge nodes gradually which is called edge computing.

Edge nodes do not have high computing power due to manufacturing cost, size and other reasons, while reducing network bandwidth consumption and latency caused by communication with data center. How to reasonably allocate computing resources for multiple tasks on an edge node or cluster of edge nodes is critical to the improvement of edge computing efficiency.

2 Background

2.1 Linux Container

Compared with the traditional virtualization technology through hardware virtualization to create a software system independent of the host, Linux Container that share the same operating system kernel and isolate the application processes from the rest of the system will save a lot of computing resources for hypervisor to emulate hardware and improve efficiency (Figure 1 [3]). In 2008, Docker came onto the scene with some new concept and technology. For example, Layers and Image version control enables developers to quickly build and deploy applications. Compared with the creation of a virtual machine that takes minutes or even hours, the creation of a new container can be completed in a few seconds.

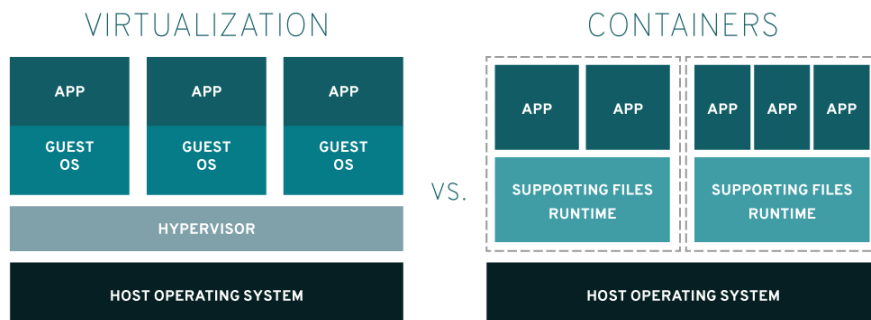


Figure 1: Virtualization vs. Containers

2.2 GPU and GPU Virtualization

CPU and GPU play completely different roles in computers. The CPU has larger caches and a more powerful arithmetic logic unit (ALU), which allows the CPU to complete complex calculations in a short time, but the parallel computing performance is not high. The GPU has smaller caches and a large number of ALU, which makes the GPU have higher performance in performing parallel computing. In other words, GPU is more suitable for performing compute-intensive tasks, such as deep learning.

Therefore, how to allocate GPU computing resources is particularly important when processing multiple deep learning tasks. Nvidia launched GRID K1 in 2013, marking the maturity of GPU virtualization technology [4]. After nearly a decade of technological development, there are many complete solutions in the cloud computing field such as Ailiyun's cGPU, Tencent's GaiaGPU[5] and Nvidia's Volta MPS[6]. But all of them focus on GPU virtualization in the cloud computing field instead of edge computing. For example, cGPU is only used for Alibaba Cloud's GPU server.

3 Design and Implementation

3.1 Design

3.1.1 Capture Video by Camera

Cameras on nodes like Jetson Nano and Raspberry Pi will record video for a given period of time, such as 10 seconds and save those video. The set of all videos from the same camera Cam_i are called Task T_i , i.e. $T_i = \{V_{i,1}, V_{i,2}, V_{i,3} \dots V_{i,m}\}$. When a new video $V_{i,m}$ from Cam_i is generated, the information about this video will be added to the Video Queue Q_i (Figure 2).

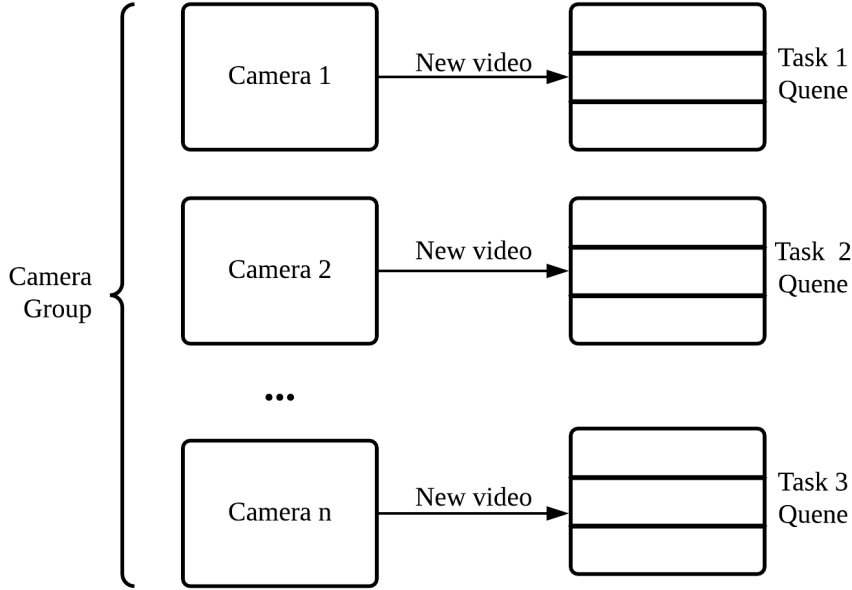


Figure 2: Cameras capture and save video

3.1.2 Assign Task to Container

The video $V_{i,top}$ at the top of each Video Queue Q_i will be put into the Task Set in manage program representing the task T_i needs to be processed if the cardinality $|Q_i| > 0$. If the container Con_f in Containers Set $\{Con_1, Con_2, \dots, Con_m\}$ is free, the video $V_{j,top}$ will be assign to this container.

In a container, when a new video is assigned, the program in the container will start to run and generate output files, which can be statistical results and rendering files for target detection. The output $O_{j,top}^f$ will be return to the manage program as the e basis to make the next allocation (Figure 3).

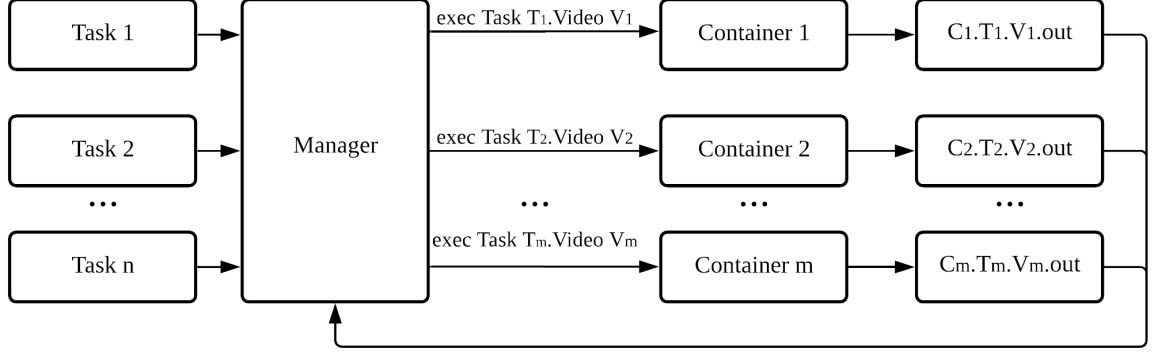


Figure 3: Assign Task to Container

Update Weights Here is a simple algorithm to update task's weights. Assume that the output $O_{j,top}^f$ of the program in container Con_f is the number of times an object has been detected. We know the total number of object in each task T_i is $Total_{obj,i}$ and a factor $k \in \mathbb{R}^+$. The new weight of task T_j

$$Total'_{obj,j} = k \times Total_{obj,j} + O_{j,top}^f$$

$$Total'_{obj} = \sum_{T_i \neq T_j} Total_{obj,i} + Total'_{obj,j}$$

$$w_j = \frac{Total'_{obj,i}}{Total'_{obj}}$$

Factor k represents the influence of the video at different times on the current weight.

$$Total_{obj,i} = O_{j,top} + kO_{j,top-1} + k^2O_{j,top-2} + \dots + k^{top}O_{j,0}$$

$k \in (0,1)$ means that the closer the video time to the current video time, the greater the influence on the current weight. $k = 1$ means all videos are equal. And $k > 1$ means that the closer the video time to the current video time, the smaller the influence on the current weight.

It can be prove that

$$\sum_{T_i} w_i = \frac{\sum_{T_i \neq T_j} Total_{obj,i} + Total'_{obj,j}}{Total'_{obj}} = 1 \quad (1)$$

After updating the weight, the manage program will check the Task Set and find a Task T_k whose proportion of the container is less than its weight w_i , assign it to the free container Con_f .

Algorithm 1: Assigning tasks to containers

Data: total number of conatiners N_{Con} , number of containers that currently executing task T_i N_{Con_i}

```
while Task Set is not empty do
  if new output  $O_{j,top}^k$  is generated then
    the weight  $w_j = \text{updateWeight}(O_{j,top}^k)$ ;
    foreach video  $V_{i,top} \in \text{Task Set}$  do
      if  $N_{Con_i}/N_{Con} < w_i$  and  $|Q_i| \neq 0$  then
        assignTask( $V_{i,top}$ ,  $Con_i$ );
        break;
      end
    end
  end
end
```

3.1.3 Example

Here we have 2 cameras(tasks) recording the videos, i.e. $Task_1$ and $Task_2$, and four container $Con_1, Con_2, Con_3, Con_4$. At first, two of them have same weight 0.5, thus each task has two containers processing. When all four containers finish processing the third video, the weights of the two tasks change from $\{0.5, 0.5\}$ to $\{0.75, 0.25\}$. When the next task is assigned, one of the containers of T_2 will execute T_1 (Figure 4).

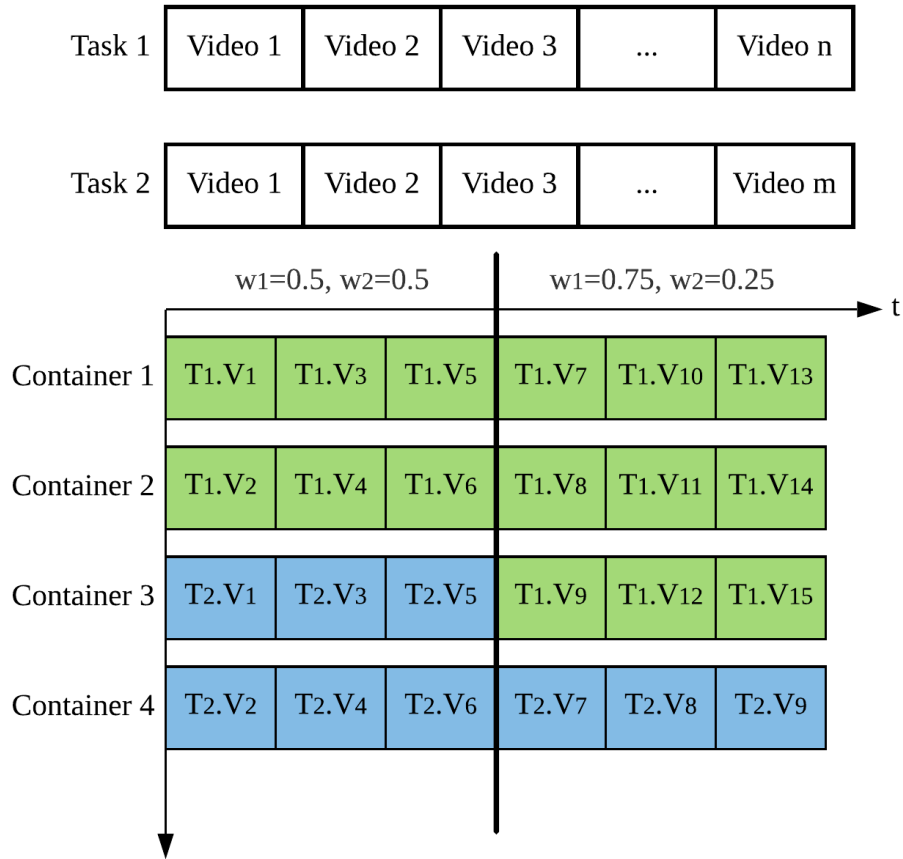


Figure 4: Example

3.2 Implementation

3.2.1 Docker Image Build

```
1 FROM python
2 RUN mkdir /code
3 COPY test.py /code
4 COPY requirements.txt /code
5 WORKDIR /code
6 RUN pip3 install -r requirements.txt
7 WORKDIR /code
8 CMD ["python3", "/code/test.py"]
```

3.2.2 Check output

```
1 import os
2 class Manager:
3     def checkOutput(self):
4         BASE_DIR = os.path.dirname(os.path.abspath(__file__))
5         outputPath=os.path.join(BASE_DIR, 'output')
6         fileList=os.listdir(outputPath)
7         fileList = sorted(fileList, key=lambda x: os.path.getmtime(os.path.join
            (outputPath, x)))
8         fileNumInLastUpdate= len(fileList)
9         with True:
10             fileList = sorted(fileList, key=lambda x: os.path.getmtime(os.path.
                join(outputPath, x)))
11             if len(fileList)>fileNumInLastUpdate:
12                 fileNumInLastUpdate+=1
13                 with open(os.path.join(outputPath, fileList[len(fileList)-1]), 'r
                    ') as f:
14                     output=f.readline()
15                     curTaskNo=int(outputName.split('-')[0])
16                     assignTask(fileList[len(fileList)-1],int(output),curTaskNo)
```

3.2.3 Update weights

```
1 class Task:
2     def __init__(self, path):
3         self.path=path
4         self.num=0
5         self.conNum=0
6         self.topVideo=None
7 class Manager:
8     def __init__(self):
9         self.k=1
10        self.tasks=[Task('1'), Task('2')]
11    def updateWeight(self, taskNo, output):
12        self.tasks[taskNo].num=self.k*self.tasks[taskNo].num+output
13    def getWeight(self, taskNo):
14        sum=0
15        for task in tasks:
16            sum+=task.num
17        return self.tasks[taskNo]/sum
```

3.2.4 Assign task

```
1 import os
2 class Container:
3     def __init__(self,cpuShares,folderPath,image):
4         self.id=os.system(" docker run -d --cpus={} --runtime nvidia -v {}:/"
5             code {} {}".format(cpuShares,folderPath,image))
6     def execTask(tak):
7         BASE_DIR = os.path.dirname(os.path.abspath(__file__))
8         inputPath=os.path.join(BASE_DIR, 'input ')
9         inputFile=os.path.join(inputPath,self.id)
10        f = open(inputFile,w)
11        f.wirte(task.topVideo)
12        f.close
13
14 class Manager:
15     def __init__(self,folderPath,image):
16         self.containers=[]
17         for i in range(2):
18             self.containers.append(Container(getWeight(i),folderPath),image)
19     def assignTask(outputName,output,curTaskNum):
20         self.tasks[curTaskNum].conNum-=1
21         curCon=outputName.split('-')[1]
22         for n in range(len(self.tasks)):
23             if self.tasks[n].conNum/len(self.containers)<self.getWeight(n):
24                 self.tasks[n].conNum+=1
25                 self.containers[curCon].execTask(self.tasks[n])
26                 break
```

4 Demo

In Figure 5, the first image shows that the output of the program in conatiner when starting. In addition, 'detach' mode is also available. The second image is an time usage result of a frame. *CPU* is the time CPU uses. *CUDA* is the the time the GPU uses. *Pre-Process* is mainly file IO such as getting the image from the video. The longest time is used by the *Network* part which uses model to process image. *Post-Process* and *Visualize* are about some computation after the Network computation like counting time-consuming, and rendering result. Figure 6 are four outputs of vehicle recognition.

```

root@nano4-desktop: /home/nano4/Desktop/Box-test
root@nano4-desktop: /home/nano4/Desktop/Box-test x jtop Nano (Developer Kit Version) - JC: Inactive - MAXN
[TRT] Registered plugin creator - ::NMS_TRT version 1
[TRT] Registered plugin creator - ::Reorg_TRT version 1
[TRT] Registered plugin creator - ::Region_TRT version 1
[TRT] Registered plugin creator - ::Clip_TRT version 1
[TRT] Registered plugin creator - ::LRReLU_TRT version 1
[TRT] Registered plugin creator - ::PriorBox_TRT version 1
[TRT] Registered plugin creator - ::Normalize_TRT version 1
[TRT] Registered plugin creator - ::RPROI_TRT version 1
[TRT] Registered plugin creator - ::BatchedNMS_TRT version 1
[TRT] Could not register plugin creator - ::FlattenConcat_TRT version 1
[TRT] Registered plugin creator - ::CropAndResize version 1
[TRT] Registered plugin creator - ::DetectionLayer_TRT version 1
[TRT] Registered plugin creator - ::Proposal version 1
[TRT] Registered plugin creator - ::ProposalLayer_TRT version 1
[TRT] Registered plugin creator - ::PyramidROIAlign_TRT version 1
[TRT] Registered plugin creator - ::ResizeNearest_TRT version 1
[TRT] Registered plugin creator - ::Split version 1
[TRT] Registered plugin creator - ::SpecialIlice_TRT version 1
[TRT] Registered plugin creator - ::InstanceNormalization_TRT version 1
[TRT] detected model format - UFF (extension '.uff')
[TRT] desired precision specified for GPU: FASTEST
[TRT] requested fastest precision for device GPU without providing valid calibrator, disabling INT8
[TRT] native precisions detected for GPU: FP32, FP16
[TRT] selecting fastest native precision for GPU: FP16
[TRT] attempting to open engine cache file networks/SSD-Mobilenet-v1/ssd_mobilenet_v1_coco.uff.1.1.710
3.GPU.FP16.engine
[TRT] loading network plan from engine cache... networks/SSD-Mobilenet-v1/ssd_mobilenet_v1_coco.uff.1.
1.7103.GPU.FP16.engine
[TRT] device GPU, loaded networks/SSD-Mobilenet-v1/ssd_mobilenet_v1_coco.uff

[TRT] -----
[TRT] Timing Report networks/SSD-Mobilenet-v1/ssd_mobilenet_v1_coco.uff
[TRT] -----
[TRT] Pre-Process CPU 0.05604ms CUDA 1.09849ms
[TRT] Network CPU 58.56255ms CUDA 52.94703ms
[TRT] Post-Process CPU 0.06406ms CUDA 0.04984ms
[TRT] Visualize CPU 0.20026ms CUDA 11.47802ms
[TRT] Total CPU 58.88292ms CUDA 65.57339ms
[TRT] -----

```

Figure 5: Demo

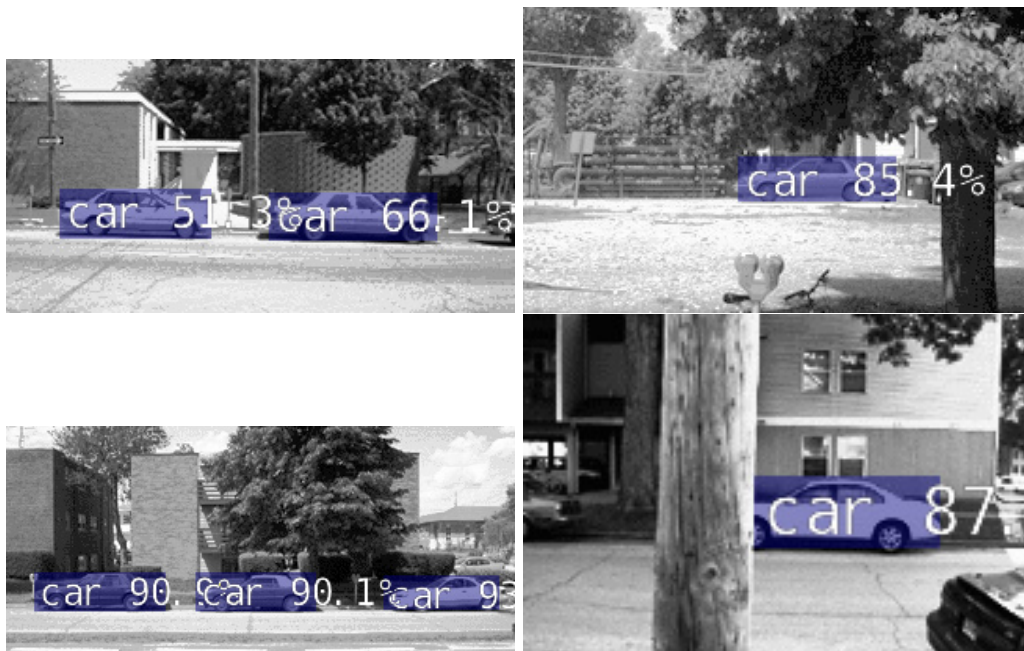


Figure 6: Output

Figure 7 shows the resource monitoring interface. The first image on the first line is the overview. The second image is the GPU usage by time without executing any task. The first image in the second line is quad-core CPU usage. The second image is the GPU usage when there are task in progress.

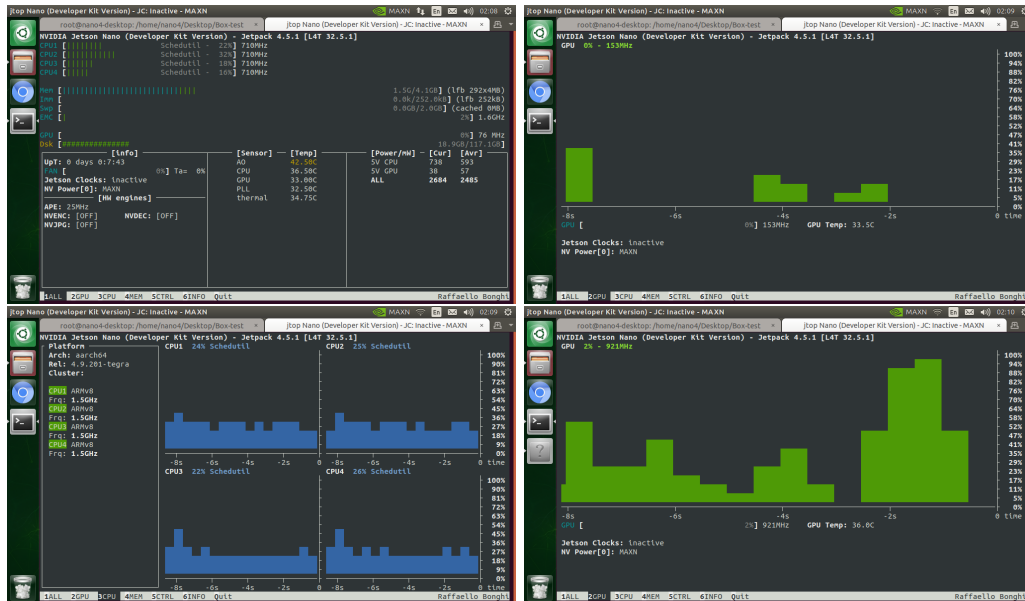


Figure 7: Resources Usage

5 Conclusion

In this report, I propose a implementation of computing power allocation. This method is different from the traditional GPU virtualization method by simulating the hardware. By dividing the task into blocks, multiple containers can execute the same task to implement the allocation of different computing power to different tasks. But due to the limitations of the Docker Engine, the smallest allocation unit is a entire card.

References

- [1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, *et al.*, “A view of cloud computing,” *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.
- [2] K. Ashton *et al.*, “That ‘internet of things’ thing,” *RFID journal*, vol. 22, no. 7, pp. 97–114, 2009.
- [3] “What’s a linux container?,” Jan 2018.
- [4] A. Herrera, “Nvidia grid: Graphics accelerated vdi with the visual performance of a workstation,” *Nvidia Corp*, pp. 1–18, 2014.
- [5] J. Gu, S. Song, Y. Li, and H. Luo, “Gaiagpu: Sharing gpus in container clouds,” in *2018 IEEE Intl Conf on Parallel Distributed Processing with Applications, Ubiquitous Computing Communications, Big Data Cloud Computing, Social Computing Networking, Sustainable Computing Communications (ISPA/IUC-C/BDCloud/SocialCom/SustainCom)*, pp. 469–476, 2018.
- [6] NVIDIA Corporation, “Multi-process service,” 2020.