Sharif University Of Technology

Computer Engineering

Operating Systems

CPU and IO Monitoring

For deep machine learning networks on huge datasets

Authors:
Alireza Ghahramani Kure,
Ahmadreza Hamzei,
Ainaz Rafiee

Professor:
Hossein Asadi

February 4, 2023

Contents

List of Figures

List of Tables

1 Abstract

The operating system lesson project is aimed at exploring the use of CPU and IO in deep learning models developed using TensorFlow, PyTorch, and OpenCV frameworks. The main steps involved in the project include bench-marking and monitoring the performance of the models on large datasets, collecting data on CPU and IO usage, and analyzing the collected data to produce charts and visualizations. The objective of the project is to gain insights into the performance of these deep learning frameworks, and to identify areas where improvements can be made. By analyzing the monitoring data and charts, the project aims to provide insights into how the use of CPU and IO affects the performance of deep learning models, and how different frameworks can be optimized for better performance.

2 Introduction

In this project, we intend to implement a number of data-oriented programs in the field of deep learning machine learning, which are widely used in heavy processing systems, by using tools of the Linux operating system in the field of I/O workload characterization, and simultaneously Check and characterization at the level of the I/O layer of the operating system.

The tools used in this project included blktrace and iostat. These two tools are executed simultaneously during the execution of applications and return very useful and accurate specifications of the input/output block layers of the operating system to the user. These specifications include the type of I/O operation, the time of the I/O operation, the addresses accessed by the application sending the request, the read & write rate per second, disk efficiency, etc.

We have implemented a number of applications in the field of deep learning using frameworks such as TensorFlow and PyTorch and libraries such as OpenCV and scikit-learn. First we run the system. Then, we perform the characterization operation on the output of two blktrace and iostat tools and finally, we display the textual or image results including the graphs and text files requested.

Here is the steps of this project, each of them will be explained in detail in the following sections.

- In the first stage, we fully implemented a number of applications from the frameworks introduced in the previous part (the extra part of the project) and then downloaded the rest of the projects suggested by the teaching assistant in the CW.
- Then we prepared the environment for the implementation of these programs. In this step, we used the virtual environment created by ”virtualenv” and installed the packages and dependencies required by the programs that are placed in ”requirements.txt” file.
- After creating the appropriate virtual environment for the components of this project, we developed tools in order to be able to monitor the requested items well. These tools are developed in python to collect the necessary information using ”blktrace” and ”iostat” then parse them to ”pandas” dataframe and draw or save the results as a csv file or plot.
- Finally, using the tools implemented in the previous section, we wrote scripts to execute the selected or implemented programs along with monitoring, and we drew the required plot susing them, .

The tools implemented with ”iostat” and ”blktrace” are explained below before other steps due to their high importance in this project.

3 IOstat Tools

3.1 Introduction

The iostat command is used for monitoring system input/output device loading by observing the time the devices are active in relation to their average transfer rates. The iostat command generates reports that can be used to change system configuration to better balance the input/output load between physical disks. A sample of report can be found in figure below: The iostat command generates three types of reports, the CPU Utilization report, the Device Utilization report and the Network Filesystem report. The command used in the project is:

$ iostat -mx sda 60

That sda is a disk to monitor and 60 is interval for reporting.

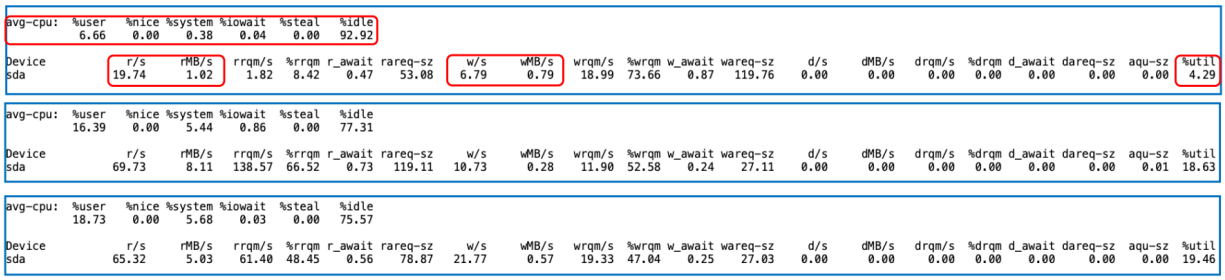

Figure 1: iostat sample report

3.2 CPU report

The first report generated by the iostat command is the CPU Utilization Report. For multiprocessor systems, the CPU values are global averages among all processors. The report has the following format:

- %user: Show the percentage of CPU utilization that occurred while executing at the user level (application).
- %nice: Show the percentage of CPU utilization that occurred while executing at the user level with nice priority.
- %system: Show the percentage of CPU utilization that occurred while executing at the system level (kernel).
- %iowait: Show the percentage of time that the CPU or CPUs were idle during which the system had an outstanding disk I/O request.
- %steal: Show the percentage of time spent in involuntary wait by the virtual CPU or CPUs while the hypervisor was servicing another virtual processor.
- %idle: Show the percentage of time that the CPU or CPUs were idle and the system did not have an outstanding disk I/O request.

3.3 Device report

The second report generated by the iostat command is the Device Utilization Report. The device report provides statistics on a per physical device or partition basis. Block devices for which statistics are to be displayed may be entered on the command line. Partitions may also be entered on the command line providing that option -x is not used. If no device nor partition is entered, then statistics are displayed for every device used by the system, and providing that the kernel maintains statistics for it. If the ALL keyword is given on the command line, then statistics are displayed for every device defined by the system, including those that have never been used. Some of fields used in the project are:

- r/s: The number of read requests that were issued to the device per second.
- rMB/s: The number of megabytes read from the device per second.
- rrqm/s: The number of read requests merged per second that were queued to the device.
- w/s: The number of write requests that were issued to the device per second.
- wMB/s: The number of megabytes written to the device per second.
- %util: Percentage of CPU time during which I/O requests were issued to the device (bandwidth utilization for the device). Device saturation occurs when this value is close to 100%.

3.4 Python parser and plotter

The iostat output is stored in text file as you can see in the shell scripts. Then the output is parsed by a python code in this repository and stores each interval as a row in csv file, two csv files are made one for cpu and another one for devices. Then the plotter code in this repository is used to extract data from files and plot theme as mentioned in project documentation. There are five outputs for the iostat tool and python code:

- iostat.txt: This file is output of running iostat command, that is executed in script. A sample is Figure 1.

- iostat_cpu.txt: The result of cpu data saved as csv format.
- iostat_device.txt: The result of device data saved as csv format.
- iostat-plot.png: The required plots in one png file.
- intensive.png: The utilization of CPU and IO is compared in one plot using data extracted using parser in csv files.

4 Blktrace-Tools

Blktrace and blkparse are tools we use to record and analyze disk I/O requests sent to storage devices. Blktrace is responsible for tracing block layer events, while blkparse interprets all the trace data collected by blktrace and provides us with graphical output and summary data.

4.1 Blktrace

To use this tool, before running the benchmark we want to monitor, we run the following command. This command starts tracking IO requests and saves the related data to the output file.

$ blktrace -d ”/dev/your disk” -a complete -o - > ”dir/trace.txt”

Then we use the following command and parse the output of the previous section in a way that can be understood and analyzed.

$ cat ”dir/trace.txt” | blkparse -i - > ”dir/parsed_trace.txt”

or

$ blkparse -i ”trace name” > ”dir/parsed_trace.txt”

4.2 Tools

In the next step, we take the parsed data as shown below and use the ”pandas” and ”matplotlib” libraries to extract the desired concepts and draw their graphs. But before we can directly use it, we have to parse it one more step to get more information.

259,0 2 1 0.000000000 0 C RM 992483336 + 8 [0]

The following code snippet performs the second step of parsing:

```
tokens = record.split()
result = {
    'timestamp': float(tokens[3]), -> 0.000000000
    'time_scope': float(tokens[3]) // time_scope_size,
    'cid': int(tokens[1]),
    'sid': int(tokens[2]),
    'pid': int(tokens[4]),
    'action': tokens[5],
    'rw': tokens[6], -> RM
    'rw_spec': 'R' if 'R' in tokens[6] else 'W' if 'W' in tokens[6] else 'N',
    'start_address': int(tokens[7]) -> 992483336
}

result.update({'n_sectors': int(tokens[9]) if '+' in tokens else 1})
result.update({'size': result['n_sectors'] * 512})
result.update({'end_address': result['start_address'] + result['size']})

result.update({
    'scope_start_address': int(result['start_address'] // address_scope_size),
    'scope_end_address': int(result['start_address'] // address_scope_size)
})
```

In the code, two variables ”time_scope_size” and ”address_scope_size” are defined to discretize addressing and nearly continuous time intervals and simplify graphing and analysis for us. For simplicity, we define ”time_scope_size” as 0.01 monitoring time and ”address_scope_size” as 25×10^6.

4.3 Dataset

Finally, the data collected by blktrace is as follows and we can call the following functions on this dataset and draw the required graphs.

Table 1: Parsed Features (1)

timestamp	time_scope	cid	sid	pid	action	rw	rw_spec	start_address
0.000000	0	2	1	0	C	RM	R	992483336
0.000313	0	2	2	0	C	RM	R	340791512
0.000373	0	2	3	0	C	RA	R	340791520
0.000395	0	2	4	0	C	RA	R	340791296
0.000486	0	2	5	0	C	RM	R	682898096

Table 2: Parsed Features (2)

n_sectors	size	end_address	scope_start_address	scope_end_address
8	4096	992487432	39	39
8	4096	340795608	13	13
40	20480	340812000	13	13
216	110592	340901888	13	13
8	4096	682902192	27	27

In this pandas dataset, the columns from left to right and from top to bottom are the time of the IO request in milliseconds from the start of monitoring, its time scope, CPU core ID, request sequence series ID on each core, action, the exact type of request, read or write request, start address and the number of sectors (each of which is 512 bytes) and request size and end address, start and end address scopes.

4.4 Blk-methods

In the following, the functions that are applied on this dataset and draw the required plots are introduced. For more information and implementation of these functions, you can refer to – its code.

```
class BLK:

    def pie_plot(self):
        ...
        plt.savefig(dir/pie_read_and_write.png)

    def density_on_size(self):
        ...
        plt.savefig(dir/density_on_size.png)

    def rw_intensive_plot(self):
        ....
        plt.savefig(dir/rw_intensive.png)

    def scope_frequency(self):
        ...
        plt.savefig(dir/address_scope_frequency.png)

    def scope_frequency_cdf(self):
        ...
        plt.savefig(dir/address_scope_freq_cdf.png)

    def hot_and_cold_scopes(self):
        ...
        plt.savefig(dir/hot_and_cold_scopes.png)
```

5 Tensorflow Applications

5.1 MNIST-classification

This code is training a deep learning model to recognize handwritten digits from the MNIST dataset. It uses the Keras library with a TensorFlow backend. The model consists of a convolutional neural network (Conv2D and MaxPooling2D layers) followed by fully-connected layers (Dense). The model is trained using categorical cross-entropy loss and the Adam optimization algorithm. The code sets a random seed to ensure reproducibility, loads the MNIST dataset, preprocesses the data (resizing and scaling), converts the labels to one-hot encoding, defines the model architecture, compiles the model, and trains it for 1 epoch with a batch size of 32. The training history is stored in the hist variable.

The code trains a deep learning model, specifically a convolutional neural network (CNN) to classify handwritten digits from the MNIST dataset. The following is a step-by-step analysis of the code:

1. Importing Libraries:
   The code starts by importing the necessary libraries. NumPy is used for numerical computing, Matplotlib is used for plotting the images, and Keras is used for building the deep learning model.

2. Setting Random Seed:
   The random seed is set using np.random.seed(123) to ensure reproducibility of results.

3. Loading MNIST dataset:
   The MNIST dataset is loaded using mnist.load_data() from Keras datasets. The dataset consists of 60,000 28x28 grayscale images of handwritten digits, with 10 classes (0 to 9). The images and their corresponding labels are split into training and test sets, which are stored in variables (X_train, y_train), (X_test, y_test).

4. Preprocessing:
   The images are reshaped to (num_samples, 28, 28, 1), and their values are normalized to be in the range [0, 1] using X_train /= 255 and X_test /= 255. This is an important step to prevent the model from being dominated by large values.

5. One-Hot Encoding Labels:
   The labels are converted from a numerical representation to one-hot encoding using to_categorical(y_train, 10) and to_categorical(y_test, 10), where 10 is the number of classes.

6. Model Architecture:
   The model is defined as a sequential model, with the following architecture:

   - Two Convolution2D layers with a relu activation function and a (3, 3) kernel. The first layer takes in the input shape of (28, 28, 1).
   - A MaxPooling2D layer with a pool size of (2, 2) to reduce the spatial dimensions of the feature maps.
   - A Dropout layer with a rate of 0.25 to prevent overfitting.
   - A Flatten layer to convert the 2D arrays to a 1D vector.
   - Two Dense layers with a relu activation function and 128 and 10 units, respectively.
   - A Dropout layer with a rate of 0.5 to prevent overfitting.

7. Model Compilation:
   The model is compiled using the categorical cross-entropy loss function, the Adam optimization algorithm, and accuracy as the metric.

8. Model Training:
   The model is trained using the model.fit function for 1 epoch with a batch size of 32. The training history is stored in the hist variable.

Regarding CPU and I/O utilization, the code will use the available CPU and I/O resources to process and load the data, define the model, and train it. The exact utilization will depend on various factors, such as the hardware specifications, the size of the dataset, and the complexity of the model. However, training deep learning models can be computationally intensive and may take a significant amount of time, especially if the dataset is large and the model is complex.

5.2 CIFAR-100-classification

The code is a python script for training a Convolutional Neural Network (CNN) on the CIFAR-100 dataset. The CIFAR-100 dataset contains 100 different classes of 32x32 color images, with 50,000 images in the training set and 10,000 images in the test set.

The code starts by importing the required libraries and modules, including TensorFlow's Keras API for building and training deep learning models. It also defines several hyperparameters for the model, such as the batch size, the loss function, the number of epochs, and the optimizer.

Next, the code loads the CIFAR-100 dataset using the load_data() function from the TensorFlow Keras datasets module and normalizes the data by dividing it by 255.

The model is then defined using the Keras Sequential model, which is a linear stack of layers. The model has 7 layers in total: 3 Conv2D layers with 32, 64, and 128 filters and a 3x3 kernel size, followed by 3 MaxPooling2D layers with a 2x2 pool size, a Flatten layer, and 2 Dense layers with 256 and 128 units respectively, with a ReLU activation function. The last layer has 100 units with a softmax activation function, as the model is being trained to classify images into 100 classes.

The model is then compiled using the loss function, optimizer, and metrics specified in the hyperparameters.

Finally, the model is trained on the input data and target labels using the fit method. The training process runs for 1 epoch and with a validation split of 0.2, meaning 20% of the training data will be used for validation and the rest for training. The script then evaluates the model's performance on the test data and prints the test loss and accuracy The code above trains a Convolutional Neural Network (CNN) model using the TensorFlow Keras library for the CIFAR-100 dataset. The CPU and I/O utilization during the training process can be impacted by several factors, including:

1. Data loading and preprocessing:
   The code loads the CIFAR-100 dataset using the cifar100.load_data() function, which can cause some I/O operations and consume CPU resources, especially if the dataset is stored on a slow storage device.
2. Model architecture:
   The model architecture consists of several Conv2D, MaxPooling2D, Flatten, and Dense layers, which can be computationally intensive and consume a lot of CPU resources.
3. Batch size:
   The batch size used during training, which is set to 50, can also impact the CPU and memory utilization. A larger batch size can cause more memory usage, but may result in a faster training process.
4. Number of epochs:
   The number of training epochs, which is set to 1, can also impact the CPU and memory utilization. More training epochs result in more computations and can cause higher CPU utilization.
5. Optimizer:
   The optimizer used to update the model's parameters, which is set to the Adam optimizer, can also impact the CPU utilization. Different optimizers have different computation requirements, and some may be faster or slower than others.

It is worth noting that the impact of these factors on the CPU and I/O utilization can depend on the specific hardware setup, as well as other factors such as the size of the dataset and the complexity of the model.

6 Pytorch Applications

6.1 MNIST-classification

The code is a script that trains a Residual Neural Network (ResNet) classifier on the MNIST handwritten digit dataset using PyTorch. It starts by defining the necessary imports and device (either CUDA GPU or CPU) to use for computations. It then performs data pre-processing on the MNIST dataset, splitting it into training and validation sets and loading it into DataLoaders. The ResidualClassifier class is then defined, which is a simple ResNet architecture with two residual blocks. A Cross Entropy Loss and SGD optimizer with an exponential learning rate schedule are then defined. The training process involves looping over a number of epochs and in each epoch, the model is set to train mode and the training data is passed through it in batches to compute the

forward pass and loss, then backpropagation is performed to update the model's parameters. In each epoch, the average training loss and accuracy are recorded. The validation process is performed similarly but with the model set to eval mode, and the average validation loss and accuracy are recorded. Finally, the script ends by plotting the recorded training and validation losses and accuracy over epochs. This code is an implementation of a residual neural network for handwritten digit recognition using the PyTorch library. The network is trained on the MNIST dataset of handwritten digits. The code utilizes CPU or GPU, depending on availability, to train the model.

In terms of CPU utilization, the code runs heavy computations during the training phase of the model. These computations include forward and backward passes through the network, as well as optimizer updates, which require a significant amount of computation on the CPU.

In terms of I/O utilization, the code downloads the MNIST dataset and saves it to the local file system. The data is then loaded into memory in batches and fed to the network during training. This means that there is some I/O utilization during the loading of the dataset, but this is typically a one-time operation and the I/O utilization during training is relatively low. Overall, the code has a relatively modest resource utilization, and will run efficiently on most modern computers, especially with GPU acceleration.

6.2 CIFAR-100-classification

This is a script for a deep learning model for image classification using the CIFAR100 dataset. It is using the PyTorch library for model building and training. The script starts with importing necessary libraries such as pandas, torch, numpy, torchvision etc.

Next, various hyperparameters for the model training are defined such as batch_size, epochs, max_lr, grad_clip, weight_decay, and the optimization function opt_func which is set to torch.optim.Adam. The CIFAR100 dataset is then loaded and a data normalization is performed to ensure that all the input data has a zero mean and unit variance. The normalization is performed by computing the mean and standard deviation of the training images and normalizing both the training and testing data with these values. The transformed data is then divided into training and testing datasets and the data loaders for both the training and testing datasets are created using torch.utils.data.DataLoader.

A function get_default_device is then defined which returns either GPU or CPU as the computing device based on the availability of GPU. A class DeviceDataLoader is defined which is used to move the data from the CPU memory to the GPU memory if GPU is available. A function accuracy is defined which computes the accuracy of the model predictions. Finally, the ImageClassificationBase class is defined which is the base class for building the deep learning model. This class contains methods for computing the training step, validation step, and the end of epoch actions. The training step computes the predictions and the cross-entropy loss for the input batch. The validation step computes the validation loss and accuracy for the input batch. The end of epoch actions are performed at the end of each epoch and print the epoch number, last learning rate, training loss, validation loss, and validation accuracy. This code is an implementation of a deep learning model for image classification using the CIFAR100 dataset. The code uses the PyTorch library for the implementation of the model and the processing of data.

The code begins with importing necessary libraries, including Pandas, Torch, Torchvision, NumPy, Matplotlib, and Scikit-learn. These libraries are used for various tasks such as data manipulation, deep learning model implementation, data processing, data visualization, and model evaluation, respectively. The batch size and number of epochs are defined as 400 and 120, respectively. Other hyperparameters, such as learning rate, gradient clipping, weight decay, and optimization function, are also defined.

The CIFAR100 dataset is then loaded, and the mean and standard deviation of the pixel values are calculated. The calculated mean and standard deviation are then used to normalize the data. The data is transformed using the 'Compose' function of the Torchvision library, which is used to apply multiple data transformations. The data is split into training and testing sets, and PyTorch's DataLoader is used to load the data in batches.

The code then implements a device agnostic data loading mechanism using the 'get_default_device' and 'to_device' functions. The 'DeviceDataLoader' class is used to wrap the data loaders and move the data to the device (GPU or CPU). The code checks if a GPU is available and selects the GPU if it is available, else it uses the CPU. The 'accuracy' function is then defined, which takes the output of the model and the ground truth labels as input and returns the accuracy of the model.

The 'ImageClassificationBase' class is then defined, which extends the PyTorch 'nn.Module' class and implements various functions for training and validation of the model. The 'training_step' function calculates the loss for each batch of the data during the training phase. The 'validation_step' function calculates the loss and accuracy for each batch of the data during the validation phase. The 'validation_epoch_end' function aggregates the losses and accuracy for all batches of the data for a single epoch during the validation phase. The 'epoch_end' function is called at the end of each epoch and is used to print the results of the epoch. Based on CPU and IO utilization, the code utilizes the CPU and IO during the data loading, data transformation, and model training phases. The GPU is utilized during the model training phase if a GPU is available. The utilization of CPU and IO resources will depend on the size of the data, the batch size, and the complexity of the model.

7 OpenCV Applications

7.1 Speed

This code is a Python script for tracking multiple objects in a video using the OpenCV and Dlib libraries. The script uses Haar cascades (carCascade = cv2.CascadeClassifier('myhaar.xml')) to detect cars in the video and then tracks their movement over time using the Dlib correlation tracker (tracker = dlib.correlation_tracker()). The function estimateSpeed takes two locations of an object and calculates its speed. The function trackMultipleObjects is the main loop of the code, where it reads each frame from the video, updates the car trackers, and detects new cars in the frame using the Haar cascades. The final result is saved to a video file ('outpy.avi'). The script also keeps track of the speed of the cars.

The code does not explicitly utilize or monitor the CPU or I/O utilization. However, it does use several CPU-intensive operations such as image processing, object detection and tracking, and writing to a video file. The code could potentially be optimized for CPU utilization by using lower-resolution images or reducing the number of object detections per frame. The I/O utilization will depend on the read and write speed of the video files, and could potentially be optimized by using faster storage devices or compressing the video files.

7.2 Yolo

This is a python script that uses the OpenCV library to perform object detection on an input image. It uses the You Only Look Once (YOLO) architecture, a deep neural network for object detection that can identify objects in real-time. The script takes in four command-line arguments: the path to the input image, the path to the YOLO configuration file, the path to the pre-trained weights for the YOLO model, and the path to the text file containing class names.

The script then loads the input image and the class names, and sets up the YOLO model using the provided weights and configuration. It processes the input image and generates detections (bounding boxes) for objects in the image. The script applies non-maximum suppression (NMS) to the detections to get the final set of detections, and draws boxes around the objects in the image along with their class labels.

The script displays the output image with object detections and saves the image to a file named "object-detection.jpg".

The CPU utilization of this code primarily depends on the size of the input image, the number of classes, the size of the model, and the number of detections performed. As the image is passed through the neural network, the CPU utilization increases. This is because the CPU is performing operations like matrix multiplications and activation functions on the input image to obtain predictions. The amount of CPU utilization can also be affected by the performance of the computer hardware and the number of CPU cores available.

The IO utilization of this code is mainly related to reading the input image, the model configuration file, the model weights file, and the class names file. The time taken for these operations is relatively small compared to the CPU utilization, as these are performed only once at the beginning of the execution. The output image is also written to disk, which will increase the IO utilization, but this operation is performed only once at the end of the execution and is therefore not a significant contributor to the overall IO utilization.

Overall, the CPU utilization is the dominant factor in terms of resource usage, as the code performs a large number of computationally intensive operations on the input image.

8 Scripts

Now we use the script defined below to perform monitoring at the same time as the application is running. This script performs all the tasks requested in the project document before running the programs and saves the reports in the specified folder.

```
# Example:
# sudo sh scripts/screens-monitoring.sh "/home/alireza/PycharmProjects/IO_monitoring/venv/bin/python"
      "src/tensorflow_apps" "cifar-100-classification-with-keras.py" "nvme0n1" "keras-classification-cifar-100"

echo none > /sys/block/nvme0n1/queue/scheduler
sync
echo 3 > /proc/sys/vm/drop_caches

mkdir -p $2/$5

screen -dmS monitoring-screen-iostat bash -c "time iostat -tx /dev/$4 1 > $2/$5/iostat.txt"
screen -dmS monitoring-screen-blktrace bash -c "time blktrace -d /dev/$4 -a complete -o - > $2/$5/trace.txt"

$1 $2/$3

screen -XS monitoring-screen-iostat quit
screen -XS monitoring-screen-blktrace quit

time cat $2/$5/trace.txt | blkparse -i - > $2/$5/parsed_trace.txt

$1 src/blktrace_monitoring/blktrace_plot.py $2/$5
$1 src/iostat_monitoring/iostat/main.py --data $2/$5/iostat.txt --disk $4 --output $2/$5/iostat.csv csv
$1 src/iostat_monitoring/iostat/main.py --data $2/$5/iostat.txt --disk $4 --fig-output $2/$5/iostat-plot.png plot
$1 src/iostat_monitoring/intensive.py $2/$5/iostat_cpu.csv $2/$5/iostat_devices.csv $2/$5

rm -f $2/$5/trace.txt

chmod -R 777 $2/$5/iostat-plot.png
chmod -R 777 $2/$5/iostat_cpu.csv
chmod -R 777 $2/$5/iostat_devices.csv
chmod -R 777 $2/$5
chmod -R 777 $2/$5/iostat.txt
chmod -R 777 $2/$5/parsed_trace.txt
```

In the script above, all the tools defined in the project are used to draw and save the required plots. In the next section, we show some of these plots. To see more examples, you can clone our code from GitHub and see more and run it if needed.

9 Sample Results

9.0.1 Tensorflow-CIFAR100-Sample

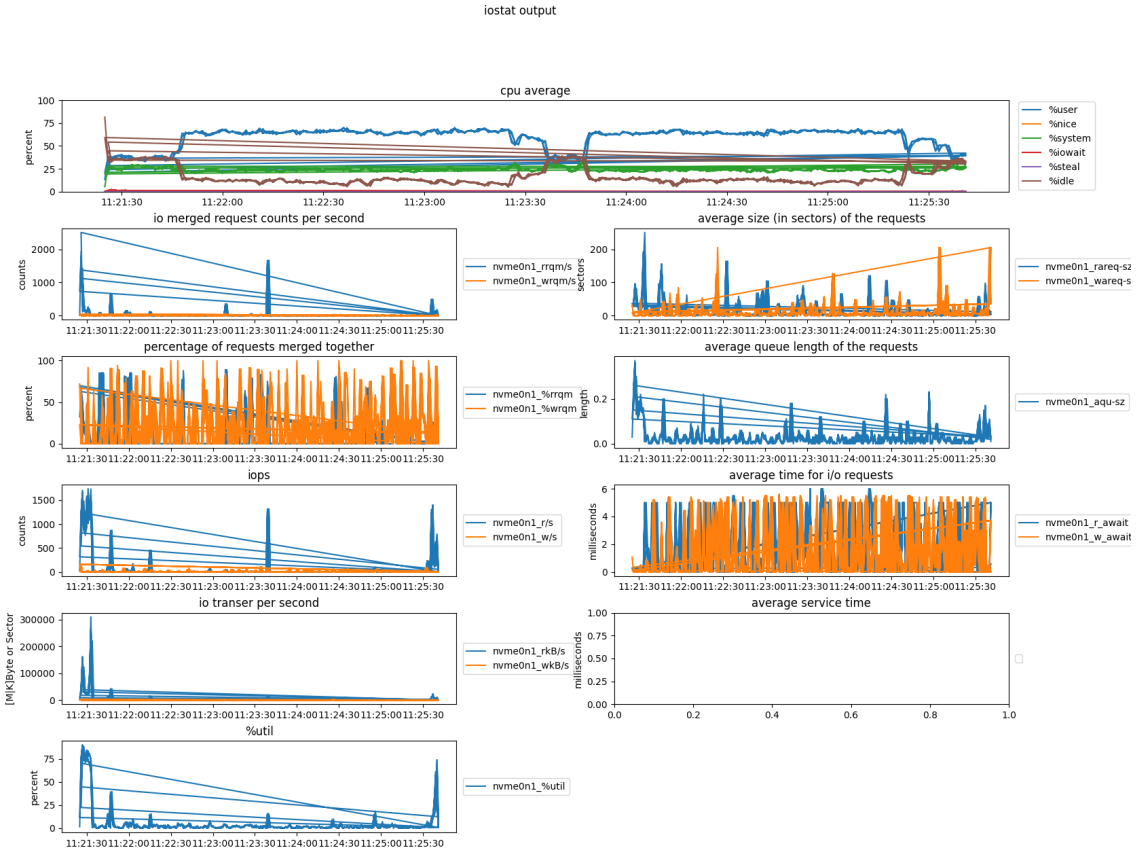

Figure 2: IOstat all plots together

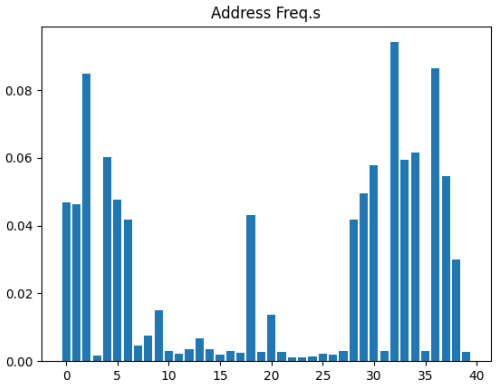

Figure 3: address scope frequecy

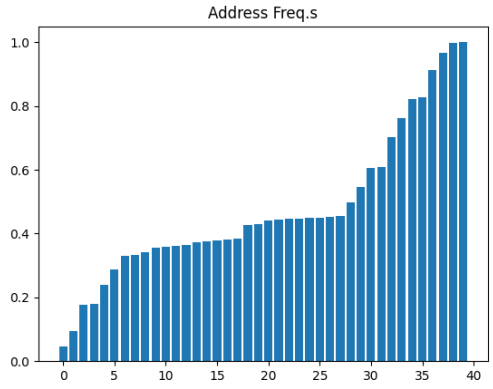

Figure 4: address scope frequecy(cdf)

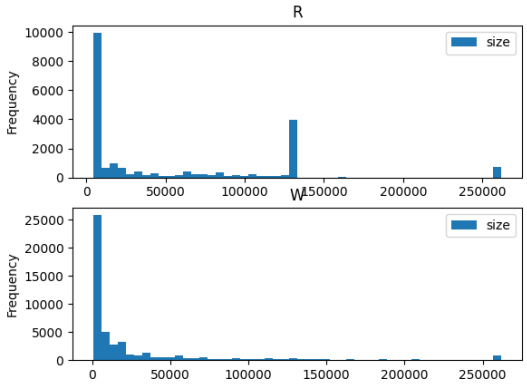

Figure 5: density of requests based on request size (R: read, W: write)

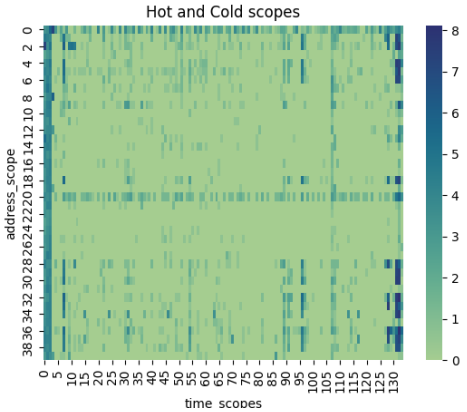

Figure 6: Hot and Cold address Scopes

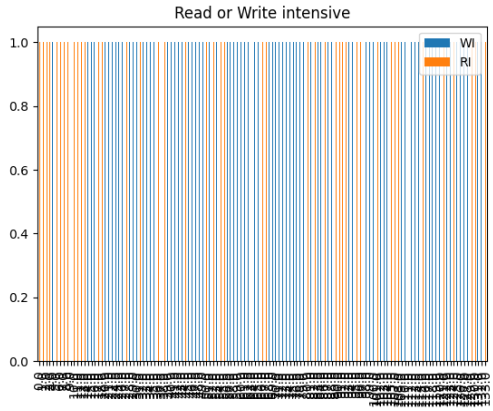

Figure 7: Read/Write intesive?

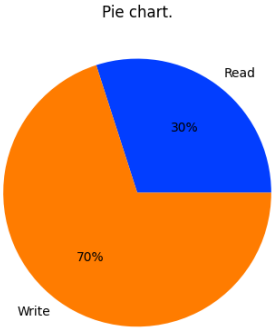

Figure 8: Read/Write pie plot

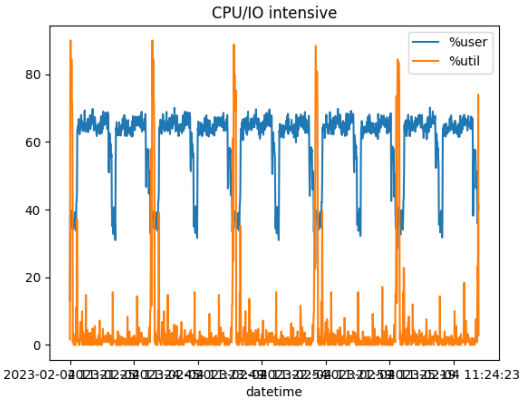

Figure 9: IO/CPU intensive plot

9.1 Pytorch-CIFAR100-Sample

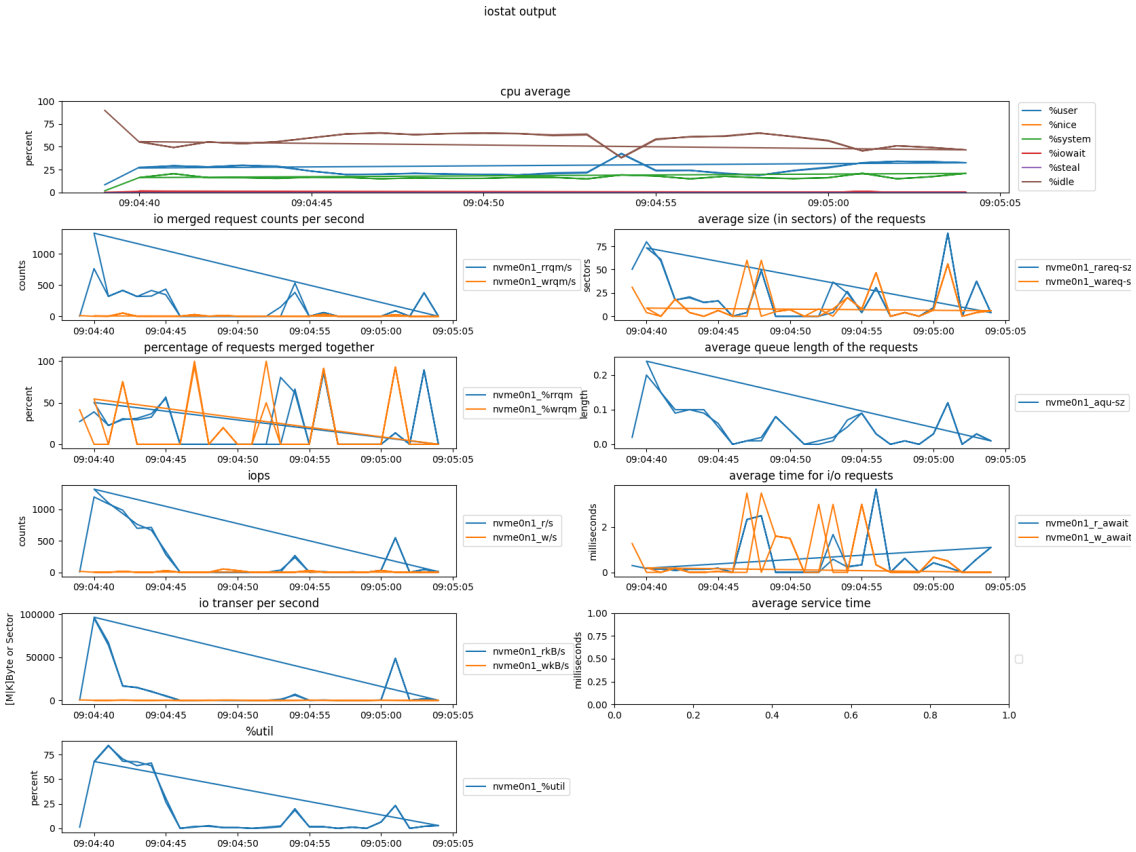

Figure 10: IOstat all plots together

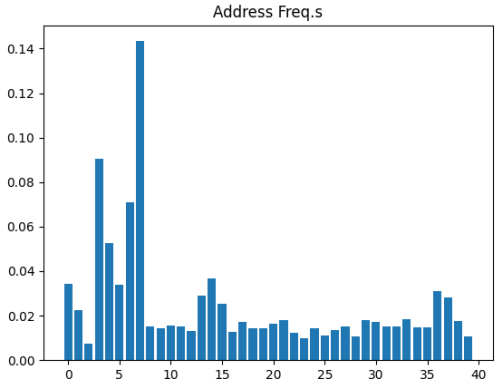

Figure 11: address scope frequecy

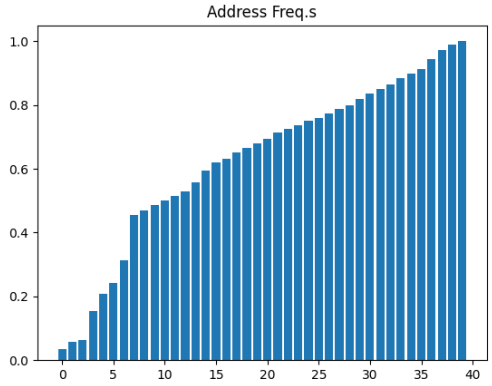

Figure 12: address scope frequecy(cdf)

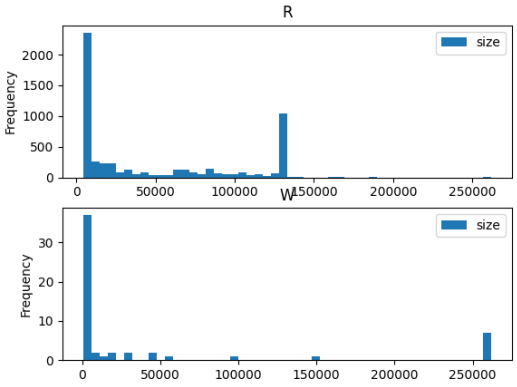

Figure 13: density of requests based on request size (R: read, W: write)

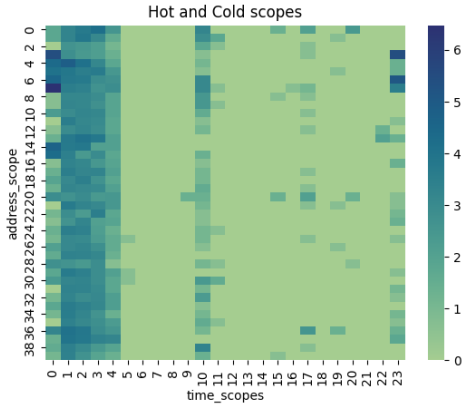

Figure 14: Hot and Cold address Scopes

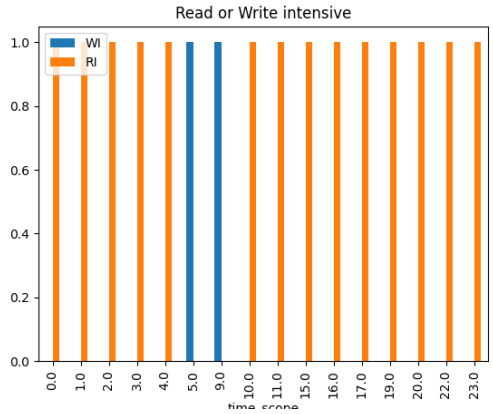

Figure 15: Read/Write intesive?

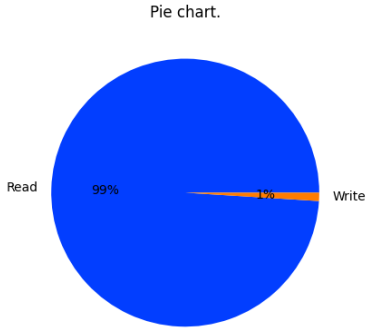

Figure 16: Read/Write pie plot

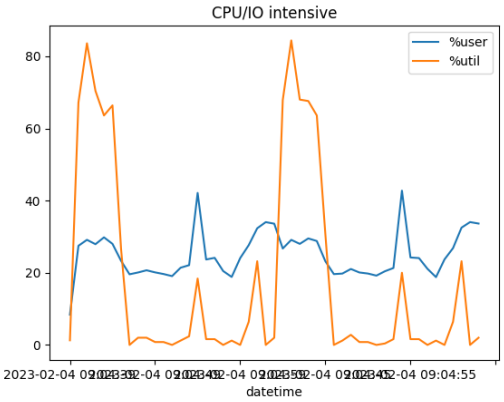

Figure 17: IO/CPU intensive plot

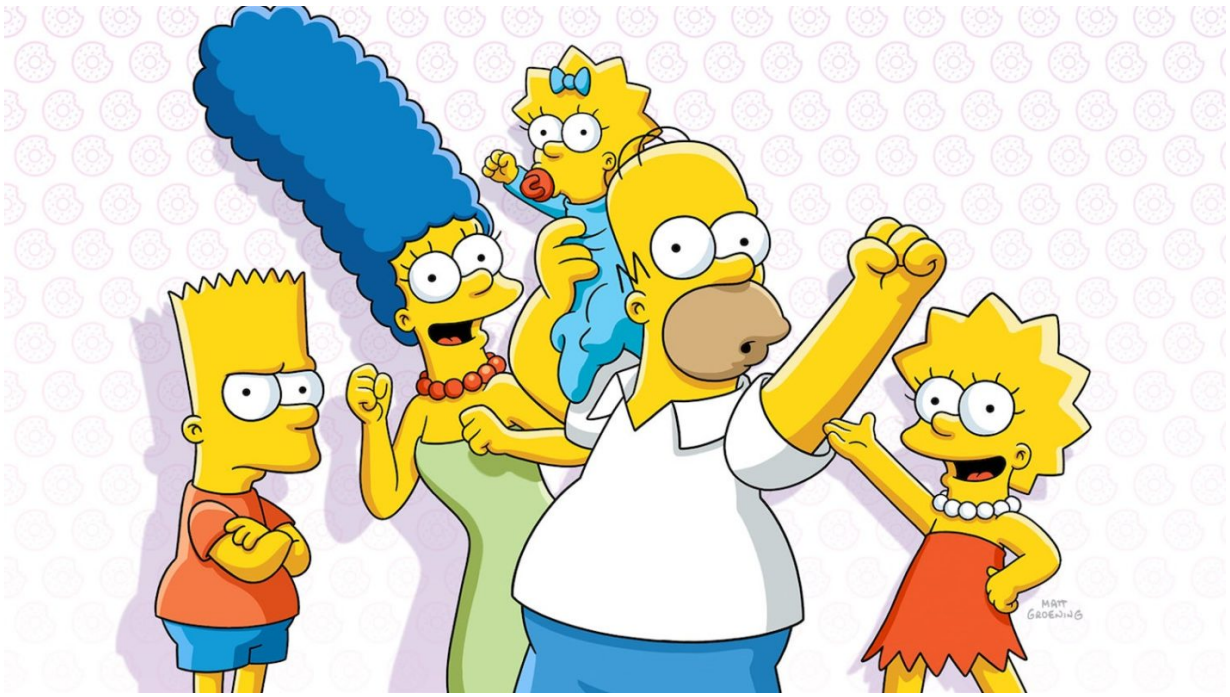

Figure 18: THE END.