



Exploratory Data Analysis

Exploring the relationships between
hardware components of a cluster server

Bash File Used to Run Experiments

```
#!/bin/bash

folder_name="$1"
benchmark_name="$2"

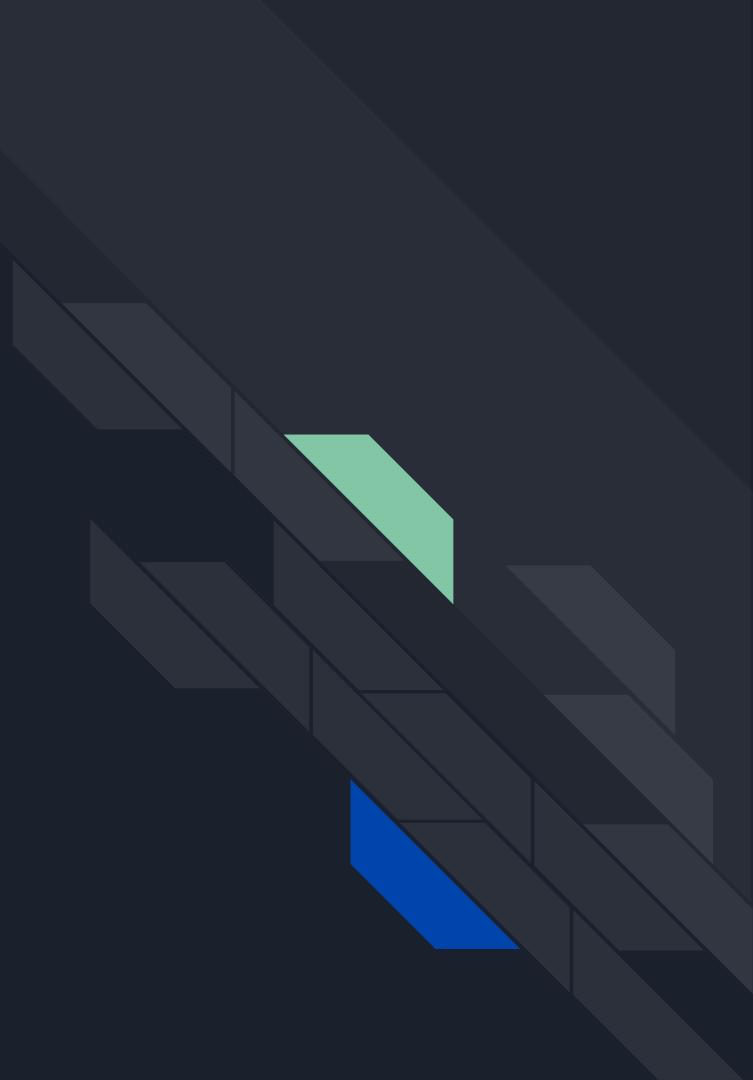
mkdir -p diagnostic_data/"$folder_name"

./cpu_util.sh > diagnostic_data/"$folder_name"/cpu_util.txt &
./cpu_temp.sh > diagnostic_data/"$folder_name"/cpu_temp.txt &
nvidia-smi -l 1 > diagnostic_data/"$folder_name"/gpu_status.txt &

if [ "$3" == "no" ]; then
    echo "Experiment: $folder_name" >> diagnostic_data/"$folder_name"/start_end_time.txt
    echo "Starts at" >> diagnostic_data/"$folder_name"/start_end_time.txt
    date >> diagnostic_data/"$folder_name"/start_end_time.txt
    python3 "$benchmark_name"
    echo "Ends at" >> diagnostic_data/"$folder_name"/start_end_time.txt
    date >> diagnostic_data/"$folder_name"/start_end_time.txt

elif [ "$3" == "yes" ]; then
    for (( i=0; i<$4; i++ )); do
        echo "Iteration $i starts at: " >> diagnostic_data/"$folder_name"/start_end_time.txt
        date >> diagnostic_data/"$folder_name"/start_end_time.txt
        python3 "$benchmark_name"
        echo "Iteration $i ends at: " >> diagnostic_data/"$folder_name"/start_end_time.txt
        date >> diagnostic_data/"$folder_name"/start_end_time.txt
        sleep 3m
    done
fi
```

Matrix Multiplication using GPU



Matrix Multiplication using GPU

Using Pytorch, create
10000 x 10000 matrices
populated by random
floating point numbers

- Send data and algorithm to GPU
- Perform matrix multiplications on the generated matrices
- Run 20 iterations
 - For our experiments we also ran this cycle 3 times, sleeping the system for 3 minutes to return to a GPU baseline temperature of 30 C
- Record Usage Data
- MatMul Operations are industry standard for GPU benchmarking

```
import torch
from tqdm import tqdm as tqdm

def run_random_matmuls(n):
    for i in tqdm(range(n)):
        tensor_1 = torch.randn(10000, 10000).to('cuda')
        tensor_2 = torch.randn(10000, 10000).to('cuda')
        torch.matmul(tensor_1, tensor_2)
        torch.cuda.synchronize()

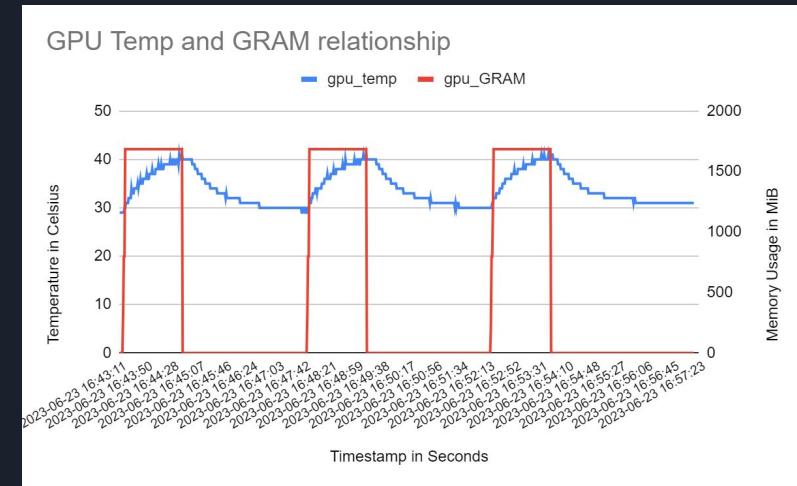
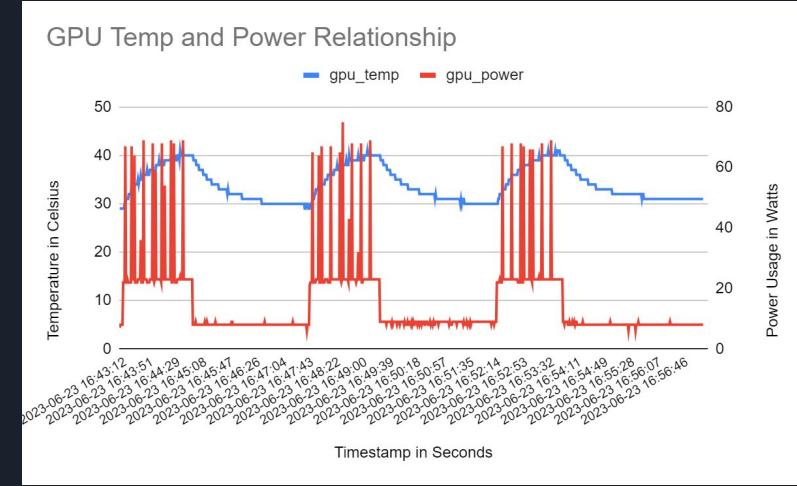
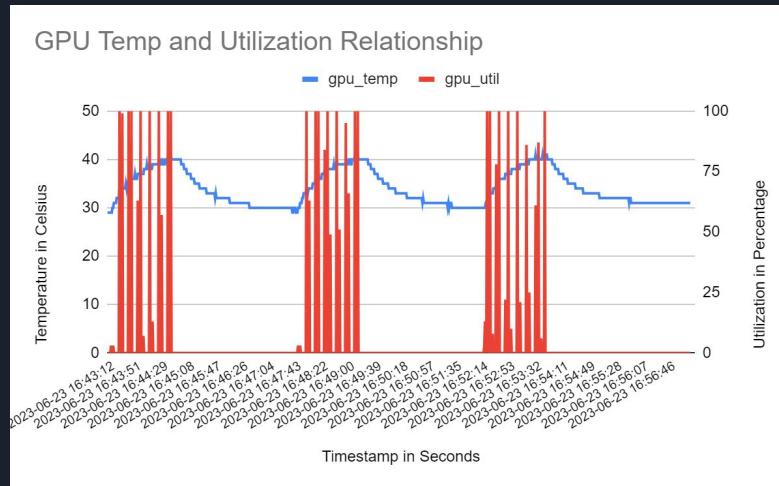
def main():
    run_random_matmuls(20)

main()
```

Matrix Multiplication using GPU

GPU Analysis

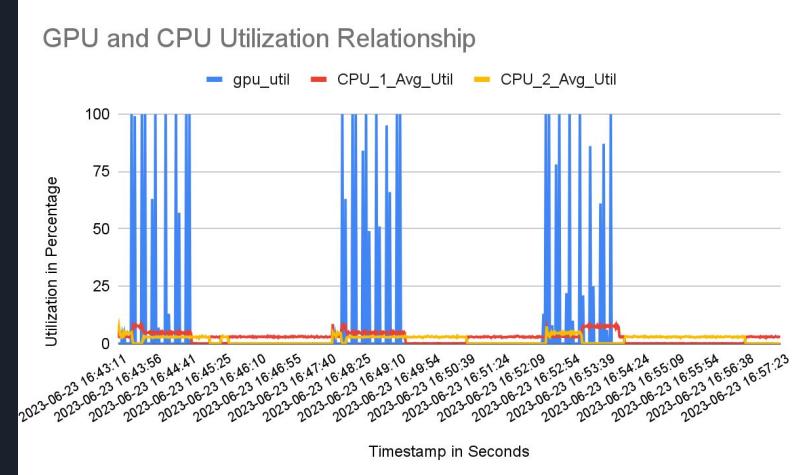
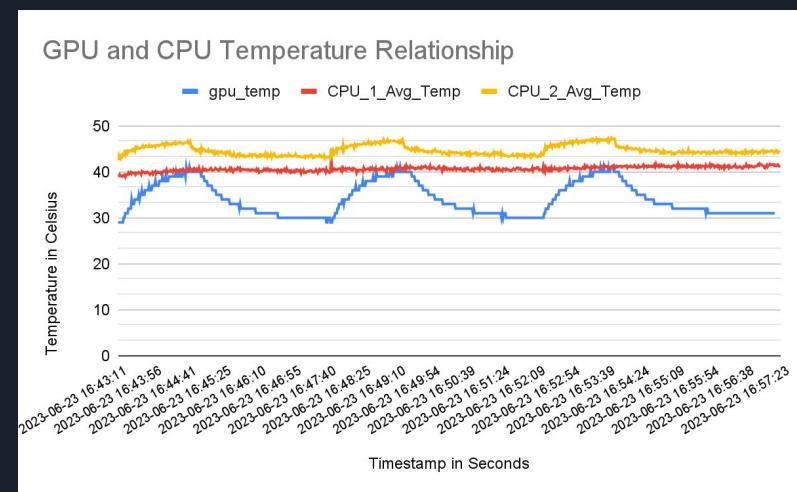
- 3 x 20 iterations of $10,000 \times 10,000$ random matrix multiplications.
- Cyclical Pattern
- Clear positive correlation between all of the GPU diagnostic data
- Peak GPU temp: 41 °C
- GPU avg util: 15% GRAM: near 30%



Matrix Multiplication using GPU

GPU and CPU Relationships

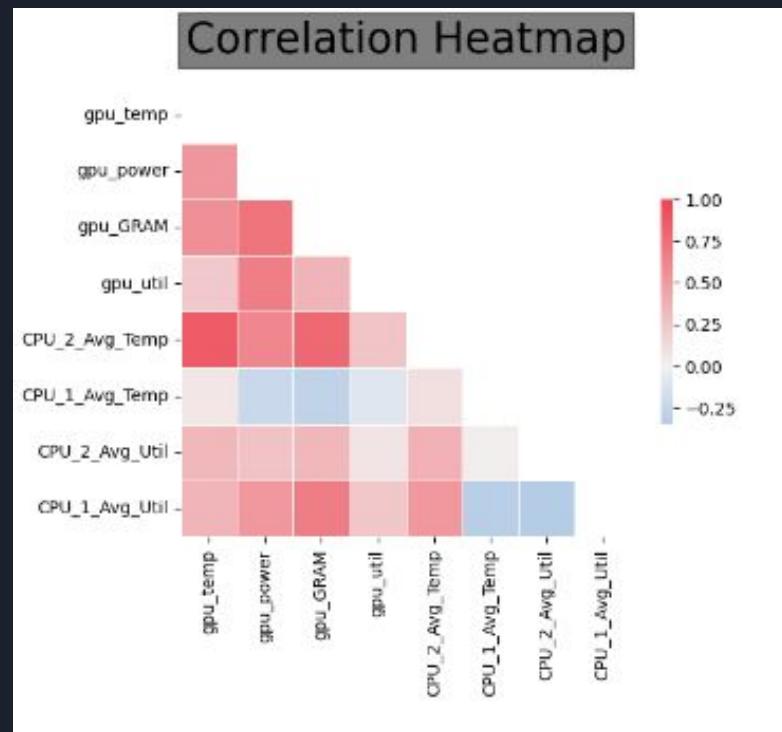
- Surprisingly, the CPU was running hotter than the GPU throughout the experiments.
 - However, Only the CPU 2 seems to be positively correlated with the GPU, and CPU 1 seems to have a slight linear relationship
- The relationship between the 2 processing units utilization is also nominal, with small CPU peaks when loading the tasks to the GPU



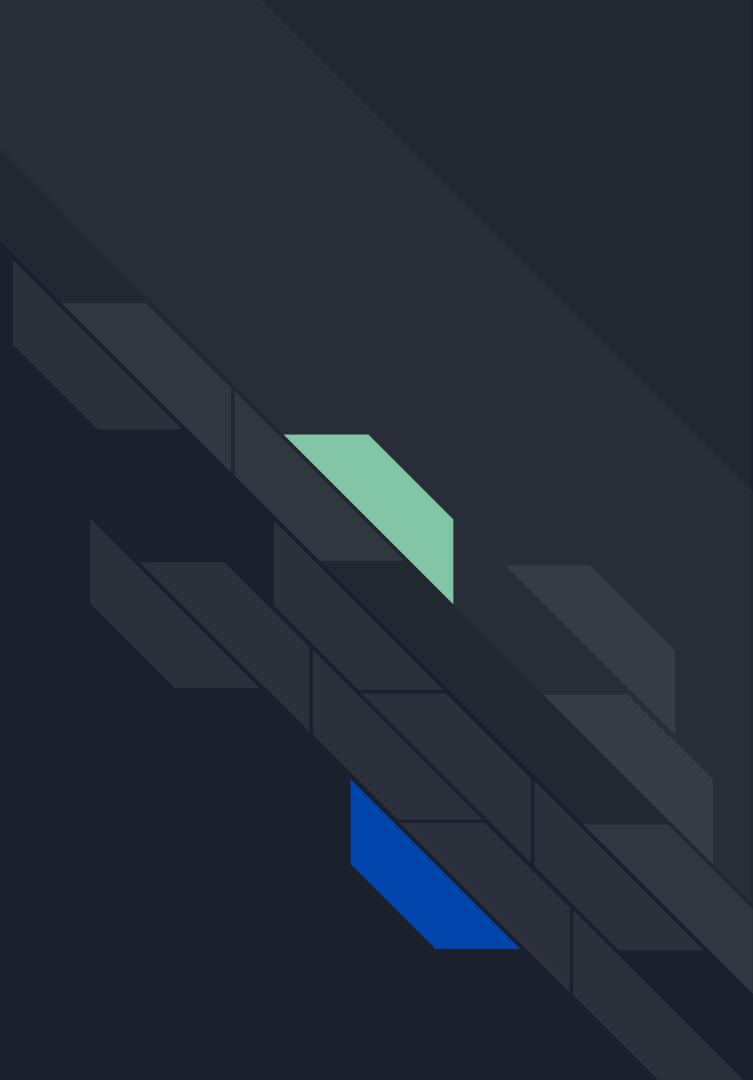
Matrix Multiplication using GPU

Feature Inter-Relationships

- Here we have a correlation plot exploring the relationships between all features and our label.
 - High values correspond to positive correlation, negative values correspond to negative correlation, and values approaching zero represent no correlation.
 - Here we can see some interesting insights, mainly that the CPU 2 temperature has a near 100% correlation with GPU temperature, whereas CPU 1 has no correlation. Perhaps, they are located in close proximity, or merely the transfer of energy and information causes the correlation.
 - Other than that, the data follows general heuristics about correlations



BERT QA LLM Prediction Task using GPU



BERT QA LLM Prediction Task using GPU

Using a BERT QA model, make predictions of answers given 5 questions and contexts

- Simulate a workload for a deployed ML model
- Using a LLM as a benchmark due to current pop culture relevance
 - Currently, LLMs take up the largest portion of current industry usage of enterprise scale GPUs for model training and deployment.
- For our experiments we ran the prediction cycle 3 times, sleeping the system for 3 minutes to return to a GPU baseline temperature of 30
- Record Usage Data and perform preliminary analysis

Unter Resentment which powered the rise of the Nazi Party, and eventually the outbreak of a Second World War in the historic palace. Most of the negotiations were in Paris, with the "Big Four" meetings taking place in the French Ministry of Foreign Affairs on the Quai d'Orsay. 1: 'Article 231, often known as the War Guilt Clause' opening article of the reparations section of the Treaty of Versailles, which ended the First World War between the German Empire and the Allied and Associated Powers. The article did not use the word "guilt" but it served as a basis to compel Germany to pay reparations for the war. \nArticle 231 was one of the most controversial parts of the treaty. It specified: \n\nThe Allied and Associated Governments affirm and Germany accepts the responsibility of Germany and her allies for causing all the loss and damage to which the Allied and Associated Governments and nations have been subjected as a consequence of the war imposed upon them by the aggression of Germany and her allies. "Germany viewed this clause as a national humiliation, forcing Germany to accept full responsibility for the war. German politicians were vocal in their opposition to the article in an attempt to generate internal sympathy, while German historians worked to undermine the article with the objective of subverting the entire article. The Allied leaders were surprised at the German reaction; they saw the article only as a necessary legal basis to extract compensation from Germany. The article, with the signatory's name changed, was also included in the peace treaty signed by Germany's allies who did not view the clause with the same disdain as the Germans did. American John Foster Dulles—one of the two authors of the article—later regretted the wording used, believing it further aggravated the German people. \n\nThe historical consensus is that responsibility or guilt for the war was not attributed to the article. Rather, the clause was a prerequisite to allow a legal basis to be laid out for the reparations that were to be made. Historians have also highlighted the unintended damage created by the clause, which increased resentment amongst the German population. 2: 'Following the ratification of article 231 of the Treaty of Versailles at the conclusion of World War I, the Central Powers were made to give war reparations to the Allies. Each of the defeated powers was required to make payments in either cash or kind. Because of the financial difficulties of Austria, Hungary, and Turkey after the war, few to no reparations were paid and the requirements for reparations were cancelled. Bulgaria, having paid only a fraction of what was required, saw its reparation figure reduced and the requirement to pay reparations removed. Historians have recognized the German requirement to pay reparations as the "chief battleground of the post-war era" and "the focus of the power struggle between France and Germany over whether the Versailles Treaty should be enforced or revised." The Treaty of Versailles (signed in 1919) and the 1921 London Schedule of Payments required Germany to pay 132 billion gold marks (US\$33 billion [all values are contemporary, unless otherwise stated] in reparations to cover civilian damage caused during the war. This figure was divided into three categories: A, B, and C. Of these, Germany was required to pay towards 'AV' and 'VB' bonds totaling 50 billion marks (US\$12.5 billion) unconditionally. The payment of the remaining 'VC' bonds was interest free and contingent on Weimar Republic's ability to pay, as was to be assessed by an Allied committee. Due to the lack of reparations payments by Germany, France occupied the Ruhr in 1923 to enforce payments, causing an international crisis. This resulted in the implementation of the Dawes Plan in 1924. This plan outlined a new payment method and raised international loans to help Germany to meet its reparation commitments. Despite this, by 1928 Germany called a new payment plan, resulting in the Young Plan that established the German reparation requirements at 112 billion gold marks (US\$26.3 billion) and created a schedule of payments that would see Germany complete payments by 1988. With the onset of the Great Depression in 1931, reparations were suspended for a year and in 1932 during the Lausanne Conference, reparations were cancelled altogether. Between 1919 and 1932, Germany paid less than 21 billion marks in reparations. Many people saw reparations as a national humiliation; the German Government worked to undermine the validity of the Treaty of Versailles and the requirement to pay. British economist John Maynard Keynes had a profound effect on historians, politicians, and the public at large. Despite Keynes' arguments and those by later historians supporting or reinforcing Keynes' view, the consensus of contemporary historians is that reparations were not as intolerable as the Germans or Keynes claimed. The capacity to pay had there been the political will to do so. Following the Second World War, West Germany took up payments. The 1953 London Agreement on German External Debts resulted in an agreement to pay 50 percent of the remaining balance. The final payment was made on 3 October 2010, settling German loan debts and reparations. 3: 'The U.S.-German Peace Treaty' was a peace treaty between the U.S. and the German government signed in Berlin on August 25, 1921. In the aftermath of World War I, the main reason for the conclusion of the peace treaty was the fact that the U.S. Senate did not consent to ratification of the multilateral peace treaty signed in Paris, thus leading to a separate peace treaty. Ratifications were exchanged in Berlin on November 11, 1921, and became effective on the same day. The treaty was registered in League of Nations Treaty Series on August 11, 1921. The assassination of Archduke Franz Ferdinand is considered one of the key events that led to World War I. Franz Ferdinand of Austria, heir presumptive to the Austro-Hungarian throne, and his wife, Sophie, Duchess of Hohenberg, were assassinated on 28 June 1914 by Bosnian Serb student Gavrilo Princip. They were shot at close range while driving through Sarajevo, the provincial capital of Bosnia-Herzegovina, formally annexed by Austria-Hungary. Princip was part of a group of six Bosnian assassins, together with Muhammed Mehmedbasic, Vasco Cubrilovic, Mihajlo Popovic, and Trifko Grabež, coordinated by Danilo Ilic; all but one were Bosnian Serbs and members of a revolutionary group that later became known as Young Bosnia. The political objective of the assassination was to precipitate the disintegration of Austria-Hungarian rule and establish a common South Slav ("Yugoslav") state. The assassination precipitated the July Crisis which led to Austria-Hungary declaring war on Serbia and the start of World War I. The assassination team was helped by the Black Hand, a Serbian secret nationalist group, support came from Dimitrijevic, chief of the military intelligence section of the Serbian general staff, as well as from Major Tankosic and Rado Malobabic, a Serbian intelligence agent. Tankosic provided bombs and pistols to the assassins in their use. The assassins were given access to the same clandestine network of safe-houses and agents.

```

import torch
from tqdm import tqdm
from tokenizers import Tokenizer
from torch.optim.lr_scheduler import LinearLR
import numpy as np
from transformers import BertConfig, AutoTokenizer, AutoModelForQuestionAnswering

class BERT_QA():
    '''class for running QA model on the summaries'''

    def __init__(self):
        '''initialize models, and variables'''

        # initialize GPU
        self.device = torch.device("cpu" if not torch.cuda.is_available() else "cuda")

        # models
        self.tokenizer = AutoTokenizer.from_pretrained("csarron/bert-base-uncased-squad-v1")
        self.model = AutoModelForQuestionAnswering.from_pretrained("csarron/bert-base-uncased-squad-v1")

        # send model to GPU
        self.model.to(self.device)

        # encoded text data, model inputs, embeddings, text
        self.encoding = {}
        self.inputs = {}
        self.sentence_embedding = {}
        self.tokens = {}

        # index of QA model's answer
        self.start_scores = {}
        self.end_scores = {}
        self.start_index = {}
        self.end_index = {}
        self.outputs = {}

        # predicted answer's text
        self.init_answer = ""
        self.cleaned_answer = ""
        self.answer_list = []

        # preprocessing variables to save split text data
        self.split_total = []
        self.split_partial = []

    def get_split_tokens(self, context):
        '''splits text from wiki summaries to be right size for BERT model'''

        if len(context.split()) // 150 > 0:
            n = len(context.split()) // 150
        else:
            n = 1
        for w in range(n):
            if w == 0:
                self.split_partial = context.split()[:200]
                self.split_total.append(" ".join(self.split_partial))
            else:
                self.split_partial = context.split()[w * 150:w * 150 + 200]
                self.split_total.append(" ".join(self.split_partial))

    def pre_process(self, question, split_wiki_summaries):
        '''tokenizes and encodes question/context for BERT model'''

        self.encoding = self.tokenizer.encode_plus(text=question, text_pair=split_wiki_summaries[0])
        # token embeddings
        self.inputs = self.encoding['input_ids']
        # segment embeddings
        self.sentence_embedding = self.encoding['token_type_ids']
        # input tokens
        self.tokens = self.tokenizer.convert_ids_to_tokens(self.inputs)

    def predict(self):
        '''uses BERT model to predict the span of text with the answer'''

        # run QA model to index predicted answer
        for batch in tqdm(range(len(self.tokens))): # Iterate over batches
            self.outputs = self.model(
                input_ids=torch.tensor([self.inputs]).to(self.device), # Use the batch input_ids
                token_type_ids=torch.tensor([self.sentence_embedding]).to(self.device) # Use the batch token_type_ids
            )
            self.start_index = torch.argmax(self.outputs.start_logits).to(self.device)
            self.end_index = torch.argmax(self.outputs.end_logits).to(self.device)

            # return predicted answer as string
            self.init_answer = " ".join(self.tokens[self.start_index: self.end_index + 1])
            for word in self.init_answer.split():
                if word[0:2] == '#':
                    self.cleaned_answer += word[2:]
                else:
                    self.cleaned_answer += ' ' + word

            # add the answer to the list
            self.answer_list.append(self.init_answer)
            self.init_answer = ""
            self.split_total = []

    qa = BERT_QA()
    questions = ["When was the treaty of versailles signed for world war one?", "What year was Archduke Franz Ferdinand assassinated?", "What was Sylvia Plath's cause of death?", "When did the Berlin wall fall?", "Where was JFK assassinated?"]

    for i in range(len(questions)):
        print(f'Device being used: {qa.device}')
        qa.get_split_tokens(contexts[i])
        qa.pre_process(questions[i], qa.split_total[0])
        print(f'Question: {questions[i]}')
        qa.predict()
        print(f'Answer: {qa.answer_list[i]}\n')

    print(qa.answer_list)

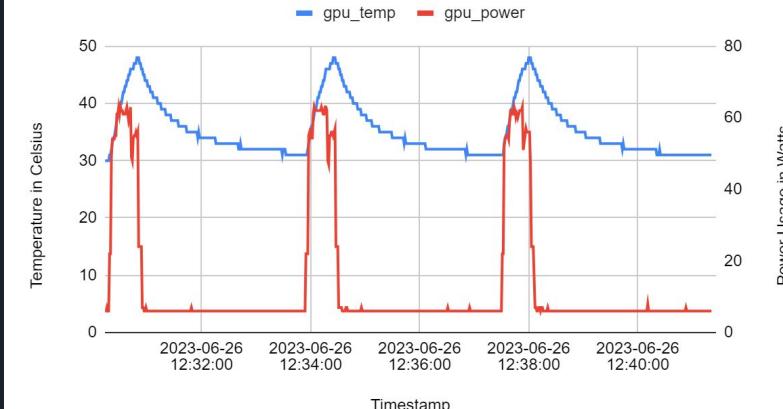
```

BERT QA LLM Prediction Task using GPU

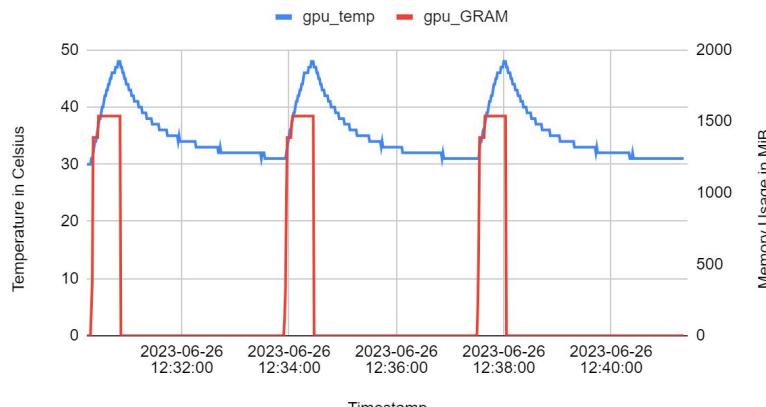
GPU Analysis

- Peak GPU temp: 48 °C
- GPU avg util: 94.37 %
- GRAM: 20%
- The Temperature and utilization behaviour differs a lot from the Matrix Multiplication experiment
 - The utilization stays at near 100% throughout each run-time
 - Temperature has a much higher peak
 - GRAM is a lot more active
- All in All, this is a great experiment for a rapid, near full utilization experiment

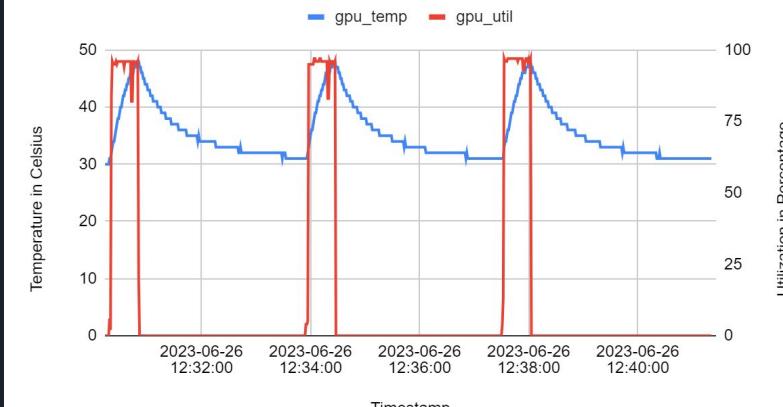
GPU Temp and Power Relationship



GPU Temp and GRAM Relationship



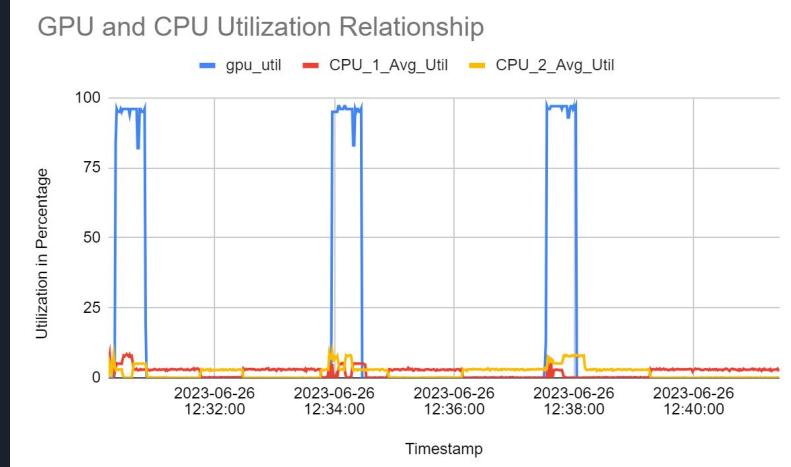
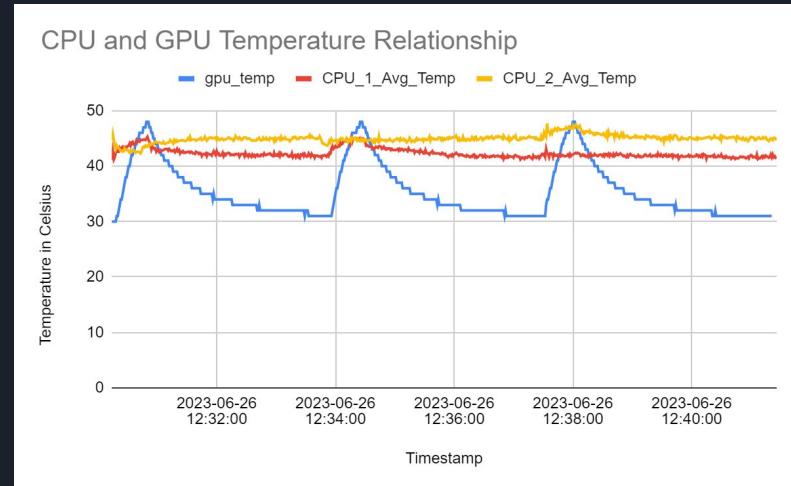
GPU Temp and Utilization Relationship



BERT QA LLM Prediction Task using GPU

GPU and CPU Relationships

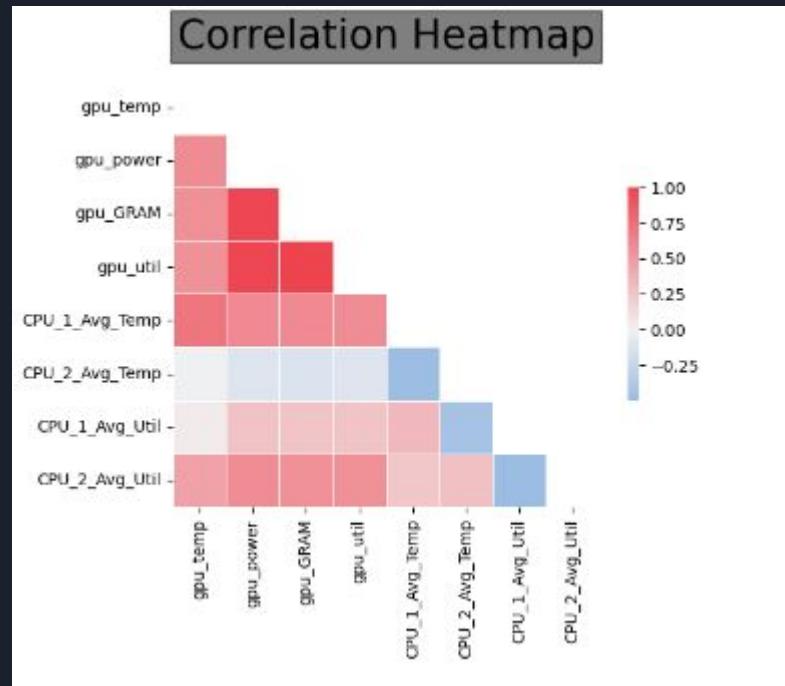
- Unlike the Matrix Multiplication experiment, it's not so obvious the direct relationship between either CPU's temperature and the GPU temperature. While there is a cause and effect, it's much more minimal than with the matmul, likely due to having the entire task loaded to GPU before running, whereas the Matrix Multiplication has a lot more interaction with the CPU
- The relationship between the 2 processing units utilization is similar to the previous experiment, with small CPU peaks when loading the tasks to the GPU



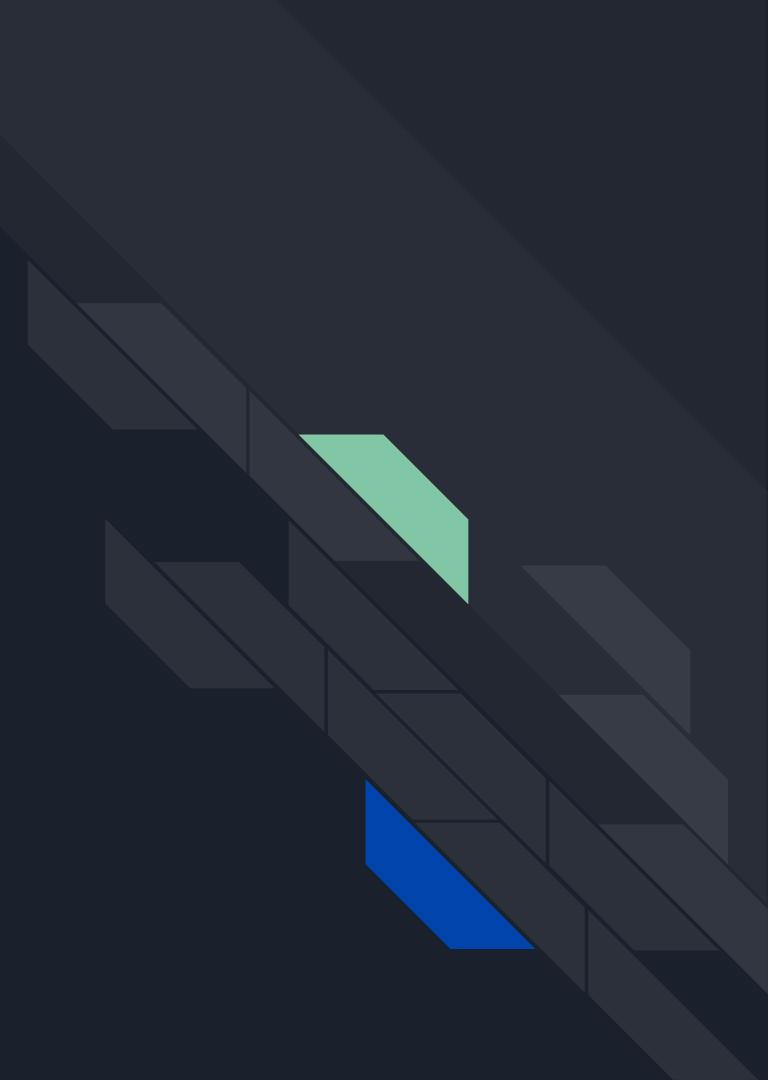
BERT QA LLM Prediction Task using GPU

Feature Inter-Relationships

- Here we have a correlation plot exploring the relationships between all features and our label.
 - High values correspond to positive correlation, negative values correspond to negative correlation, and values approaching zero represent no correlation.
 - Most of the correlations remain the same for this experiment except for the CPU temperature. While CPU 1 temperature does have a positive correlation with the GPU temperature, CPU 1 utilization has low-negative to no correlation with our label.
 - Other than that oddity, the data follows general heuristics about correlations between hardware states



distilBERT QA LLM
Training Task using GPU



distilBERT QA LLM Training Task using GPU

Using a distilBERT base model, run one epoch of training using the SQuAD dataset to finetune for QA

- Simulate a workload for a training a large ML model
- Using a LLM as a benchmark due to current pop culture relevance
 - The highest barrier to entry for training cutting edge language models is the extraordinary expensive cooling costs that accompany such intensive programs
- SQuAD dataset is a dataset for QA created by Stanford researchers and made public for training and benchmarking LLMs
- For our experiments we ran one training epoch which took approximately 1.5 hours on our Nvidia Tesla GPU
- Record Usage Data and perform preliminary analysis

```
{  
  "data": [  
    {  
      "paragraphs": [  
        {  
          "context": "The Normans (Norman: Nourmands; French: Normands; Latin: Normanni) were the people who in the 10th and 11th centuries gave their name to Normandy, a region in France. ",  
          "qas": [  
            {  
              "answers": [  
                {  
                  "answer_start": 159,  
                  "text": "France"  
                }  
              ],  
              "id": "56ddde6b9a695914005b9628",  
              "is_impossible": false,  
              "question": "In what country is Normandy located?"  
            }  
          ]  
        },  
        {  
          "title": "Normans"  
        }  
      ],  
      "version": 2  
    }  
  ]  
}
```

Figure shows the general format of the SQuAD dataset, in JSON format
Source: <https://rajpurkar.github.io/SQuAD-explorer/>

```

# !pip install datasets
import warnings # to ignore tqdm warning (irrelevant/ annoying warning)
warnings.filterwarnings("ignore") # set parameter to ignore warnings
import datasets # to get datasets from huggingface's api
from datasets import load_dataset # for loading in those datasets
from tqdm.auto import tqdm # for progress bar
from transformers import DistilBertTokenizerFast # BERT tokenizer to get encodings
import torch # NV Framework
from transformers import AdamW # Optimizer function from huggingface
from transformers import DistilBertForQuestionAnswering # QA model

def add_end_idx(answers, contexts):
    new_answers = []
    # loop through all of the answer-context pairs
    for answer, context in tqdm(zip(answers, contexts)):
        # reformatted to remove lists
        answer['text'] = answer['text'][0]
        answer['answer_start'] = answer['answer_start'][0]
        # gold_text is the answer we are looking to find in the context
        gold_text = answer['text']
        # assign start index for answer span and end index with len of answer span
        start_idx = answer['answer_start']
        end_idx = start_idx + len(gold_text)

        # accounting for squad dataset not always being correct with answer span indices
        if context[start_idx:end_idx] == gold_text:
            # if the indexing is correct
            answer['answer_end'] = end_idx
        else:
            # if the answer index length is off by a couple of tokens, check for matches in near indices
            for n in [1, 2]:
                #+- the answer span gets saved
                if context[start_idx - n: end_idx - n] == gold_text:
                    answer['answer_start'] = start_idx - n
                    answer['answer_end'] = end_idx - n

        # append either answer span to a list
        new_answers.append(answer)
    return new_answers

def prep_data(dataset):
    # create dictionary out of q/a/context groupings
    questions = dataset['question']
    contexts = dataset['context']
    answers = add_end_idx(dataset['answers'], contexts)

    return {'question': questions, 'context': contexts, 'answers': answers}

def add_token_positions(encodings, answers, tokenizer):
    # initialize lists to hold start/end indicies of tokenized answer spans
    start_positions = []
    end_positions = []

    for i in tqdm(range(len(answers))):
        # append start/ end token position using char_to_token
        start_positions.append(encodings.char_to_token(i, answers[i]['answer_start']))
        end_positions.append(encodings.char_to_token(i, answers[i]['answer_end']))

        # if start position is Nonetype, the anwer has been truncated
        if start_positions[-1] is None:
            start_positions[-1] = tokenizer.model_max_length
        # end position cannot be found so shift the index
        shift = 1
        while end_positions[-1] is None:
            end_positions[-1] = encodings.char_to_token(i, answers[i]['answer_end']) - shift
            shift += 1
    # update encodings object with new start/end answer spans
    encodings.update({'start_positions': start_positions, 'end_positions': end_positions})

class SquadDataset(torch.utils.data.Dataset):
    def __init__(self, encodings):
        # Initialize the dataset with the provided encodings
        self.encodings = encodings

    def __getitem__(self, idx):
        item = {}
        for key, val in self.encodings.items():
            if isinstance(val[idx], torch.Tensor) and val[idx].dtype.is_floating_point:
                item[key] = val[idx].clone().detach().requires_grad_(True)
            else:
                item[key] = val[idx]
        return item

    def __len__(self):
        # Return the length of the dataset, which is determined by the number of input_ids
        return len(self.encodings.input_ids)

def main():
    # Load in and format dataset for QA
    # Disable annoying progress bar
    # datasets.disable_progress_bar()
    # Load the dataset from the 'oscars' dataset in the 'unshuffled_deduplicated_en' split
    dataset = datasets.load_dataset('oscars', 'unshuffled_deduplicated_en', split='train', streaming=True)
    # Load the SQuAD dataset
    data = load_dataset('squad')
    # Preprocess the SQuAD dataset
    dataset = prep_data(data['train'])

    # Load the pre-trained tokenizer for BERT from Hugging Face
    # tokenzier = DistilBertTokenizerFast.from_pretrained('distilbert-base-uncased')
    # Tokenize the dataset
    train = tokenizer(dataset['context'], dataset['question'], truncation=True, padding='max_length', max_length=512, return_tensors='pt')

    # Create token-based start/end indices
    add_token_positions(train, dataset['answers'], tokenizer)

    # Build the dataset using the SquadDataset class
    train_dataset = SquadDataset(train)

    # Create a data loader for batch processing
    loader = torch.utils.data.DataLoader(train_dataset, batch_size=16, shuffle=True)
    device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
    # Load the DistilBERT model for question Answering
    model = DistilBertForQuestionAnswering.from_pretrained('distilbert-base-uncased')
    model.to(device)
    model.train()
    optim = torch.optim.AdamW(model.parameters(), lr=5e-5)

    # Training loop
    for epoch in range(1):
        loss = 0
        for batch in loader:
            batch = {k: v.to(device) for k, v in batch.items()}
            start_positions = batch['start_positions'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            start_positions = batch['start_positions'].to(device)
            end_positions = batch['end_positions'].to(device)

            # Clear any previously calculated gradients
            optim.zero_grad()

            # Move the input tensors to the appropriate device (CPU or GPU)
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            start_positions = batch['start_positions'].to(device)
            end_positions = batch['end_positions'].to(device)

            # Clear any previously calculated gradients
            optim.zero_grad()

            # Forward pass through the model
            outputs = model(input_ids, attention_mask=attention_mask, start_positions=start_positions, end_positions=end_positions)

            # Compute the loss
            loss = outputs.loss

            # Perform backpropagation and update the model's parameters
            loss.backward()
            optim.step()

            # Update the progress bar description and display the current loss
            loop.set_description(f'Epoch {epoch}')
            loop.set_postfix(loss=loss.item())

    # Save the trained model
    model.save_pretrained('./distilbert-qa-test')

main()

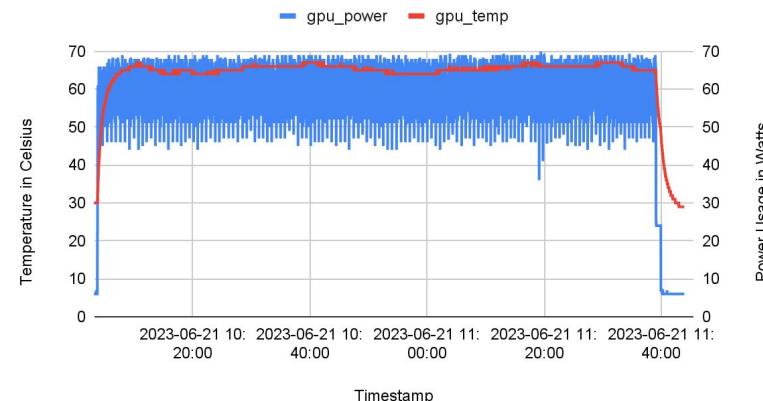
```

distilBERT QA LLM Training Task - GPU Analysis

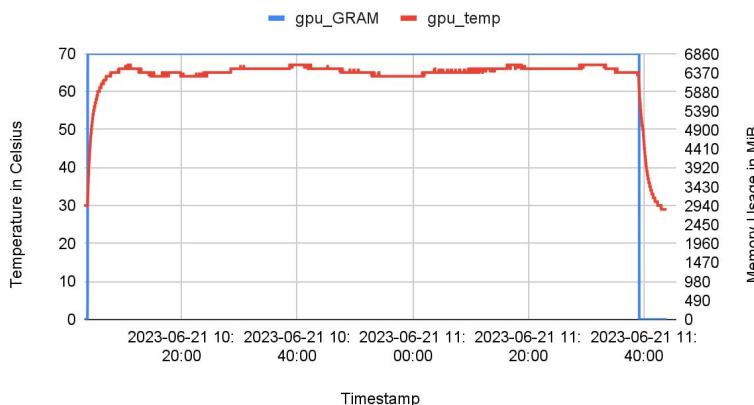
- Peak GPU temp: 67 °C
- GPU avg util: 99.62 %
- GRAM: 90.5%

- This experiment is as power, temperature, and utilization intensive as possible
 - The utilization stays 100% for practically the entire runtime of the training session
 - Temperature stays at a peak range between 68 and 63 for the entire experiment
 - Rapid power fluctuations between 55 and 70 watts through the whole experiment
 - Full memory usage throughout the experiment
- This experiment is useful to analyze and simulate the most GPU intensive tasks being performed on an enterprise scale daily

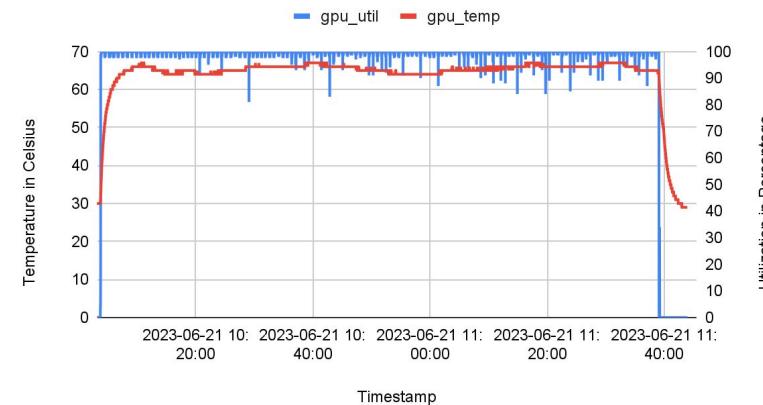
GPU Temp and Power Relationship



GPU Temp and GRAM Relationship

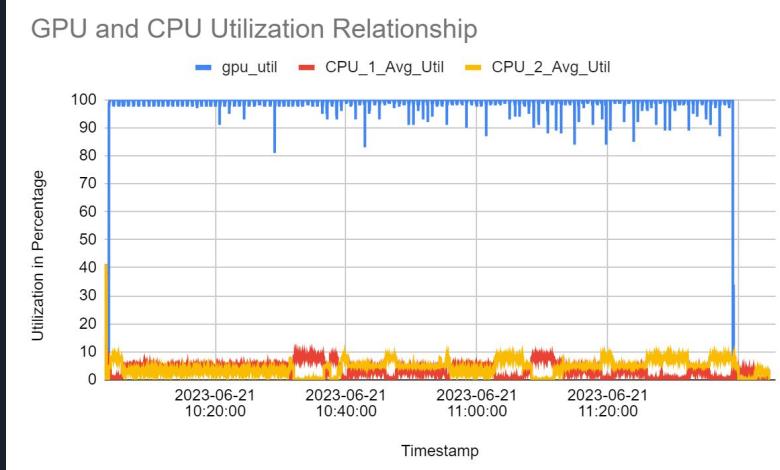
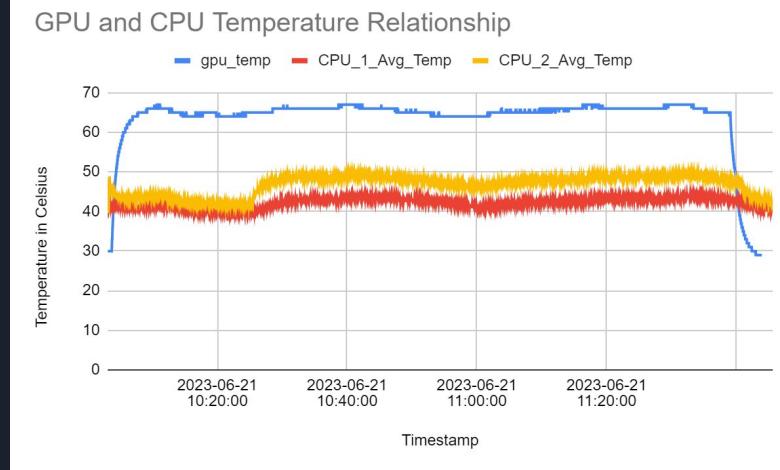


GPU Temp and Utilization Relationship



GPU and CPU Relationships

- There appears to be minimal correlation between the GPU and CPU temperatures and Utilization
 - This checks out with the heuristic that a fully intensive GPU program would not have many interactions at all with the CPU
- The main anomaly would be the temperature spike of each CPU about $\frac{1}{4}$ of the way through the experiment.
 - No current heuristic for understanding what is happening



distilBERT QA LLM Training Task using GPU

Feature Inter-Relationships

- Here we have a correlation plot exploring the relationships between all features and our label.
 - High values correspond to positive correlation, negative values correspond to negative correlation, and values approaching zero represent no correlation.
 - As this program is the most GPU intensive of all of our experiments, it makes sense that there is minimal correlation between the CPU and GPU behaviour, although the increase in environmental temperature is due to have a compounding effect.
 - The main oddity here would be how the CPU 1 utilization rate has no bearing on GPU temperature, although the temperature does, albeit minimally. However, this follows the notion that having more active hardware increases the general heat of the cluster server

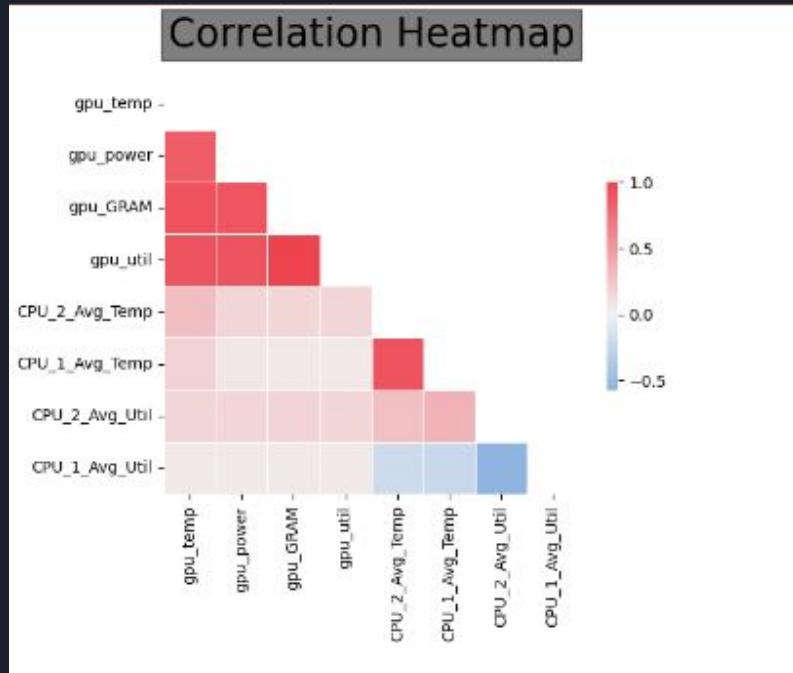


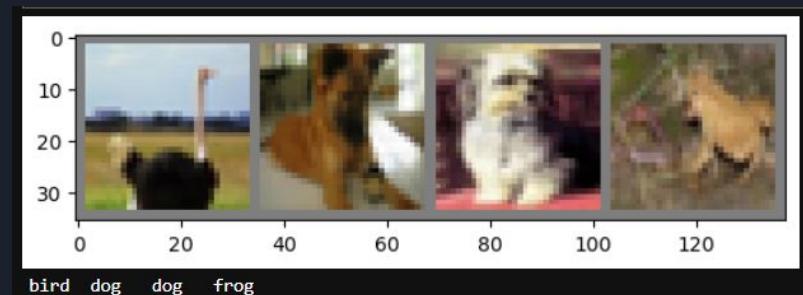
Image Classification with Pytorch on GPU



Image Classification with Pytorch on GPU

Using a torchvision image classifier to predict predetermined label types for images

- Simulate a workload for training an image NN
- Computer Vision and Image Classification are both very popular use cases for ML and consequently, GPU usage
- The dataset is provided by Pytorch for learning image classification, and contains only 10 different potential labels
- For our experiments we ran 2 training epochs which took approximately 10 minutes on our Nvidia Tesla GPU
- Record Usage Data and perform preliminary analysis



```
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

Figures show the general format of this dataset

```

import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from tqdm import tqdm as tqdm

transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

batch_size = 4

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                         download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

net = Net()
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
net.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

for epoch in range(2): # loop over the dataset multiple times

    running_loss = 0
    for i, data in tqdm(enumerate(trainloader, 0)):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data[0].to(device), data[1].to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999: # print every 2000 mini-batches
            print(f'[epoch {epoch+1}, {i+1:5d}] loss: {running_loss / 2000:.3f}')
            running_loss = 0

    dataiter = iter(testloader)
    images, labels = next(dataiter)

    ##### Testing the model
    images, labels = images.to(device), labels.to(device)
    outputs = net(images)

    _, predicted = torch.max(outputs, 1)
    print('Predicted: ', ' '.join(f'{classes[predicted[j]]:5s}'
                                  for j in range(4)))

    correct = 0
    total = 0
    # since we're not training, we don't need to calculate the gradients for our outputs
    with torch.no_grad():
        for data in testloader:
            images, labels = data[0].to(device), data[1].to(device)
            # calculate outputs by running images through the network
            outputs = net(images)
            # the class with the highest energy is what we choose as prediction
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    print(f'Accuracy of the network on the 10000 test images: {100 * correct // total} %')

```

Image Classification with Pytorch on GPU

- Right away, we can notice a much lower load on the GPU despite being an intensive task
 - For example, utilization peaks at 21 %
- Generally, this is a more computationally efficient model than the LLMs, and provides an example of low GPU power, memory, and utilization raising the GPU temp to a non-trivial amount

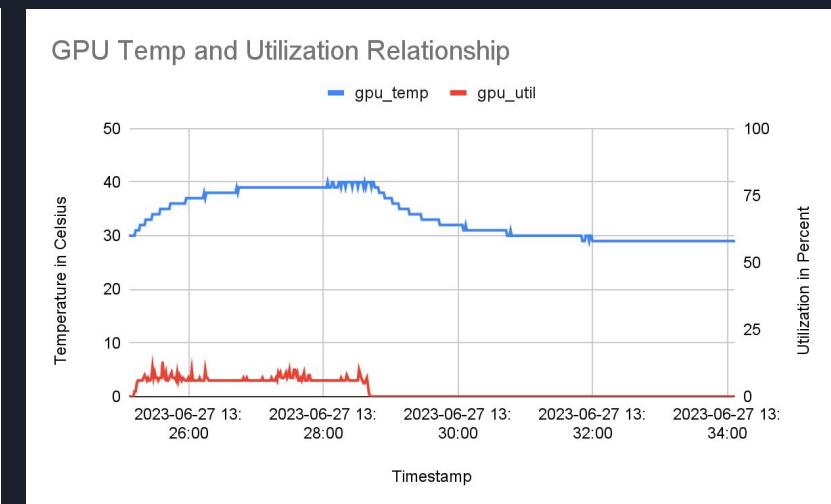
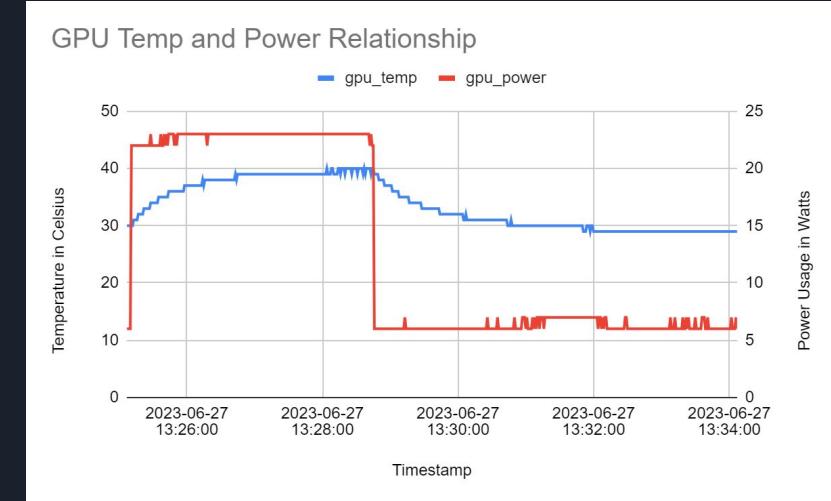
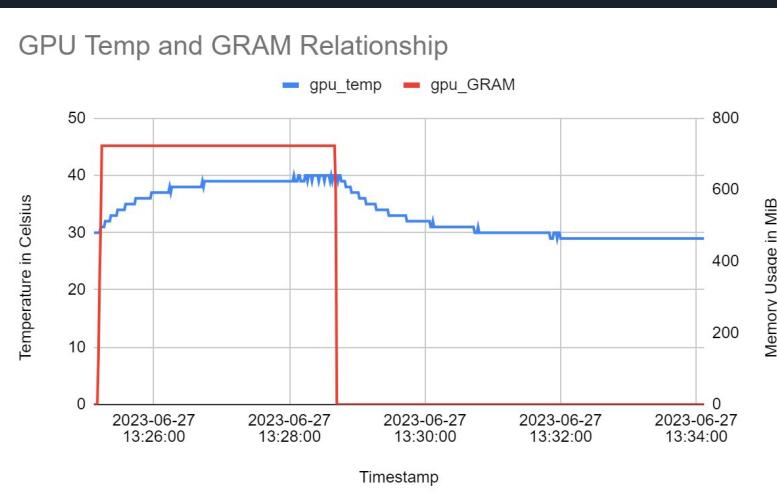
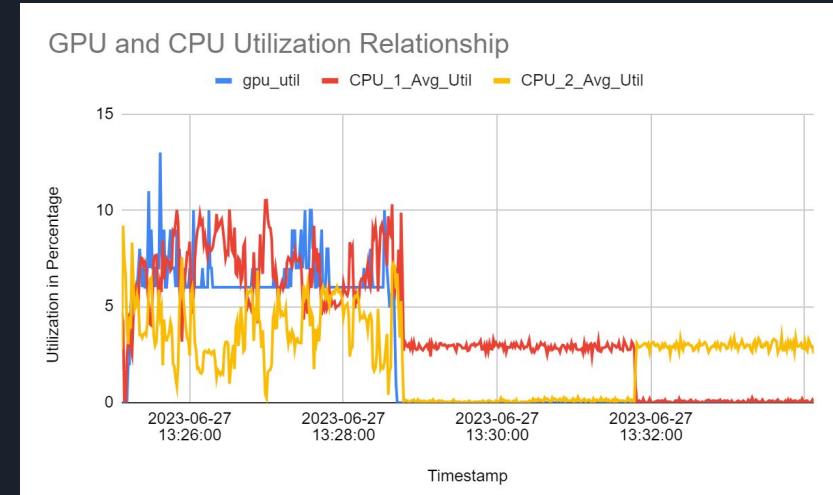
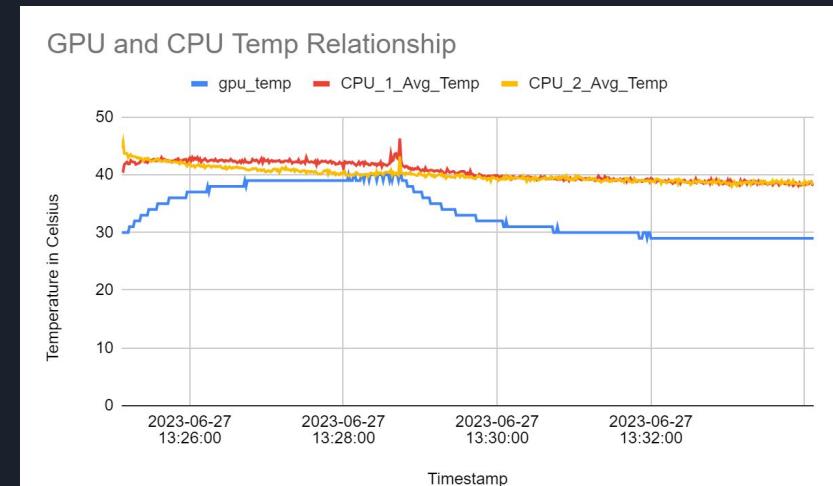


Image Classification with Pytorch on GPU

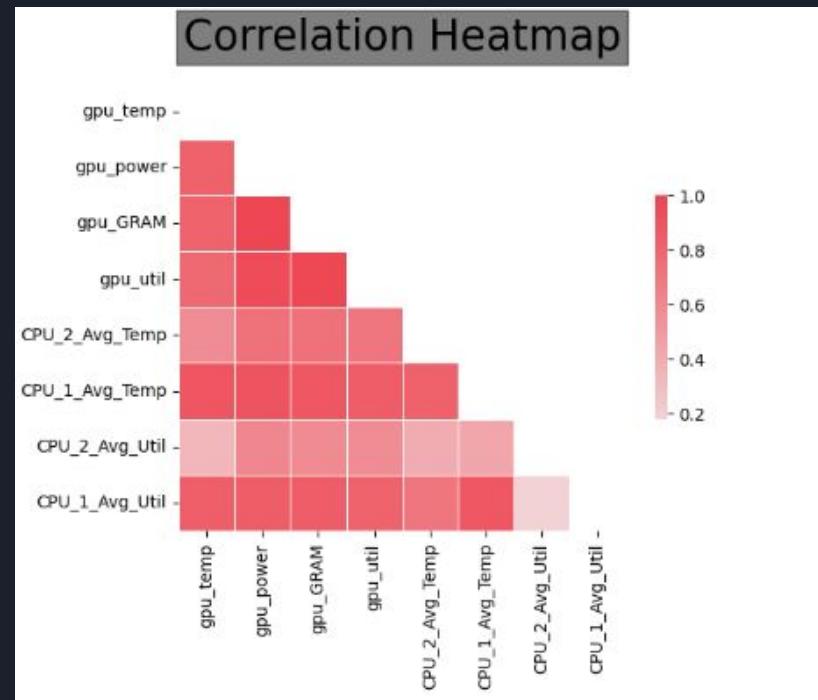
GPU and CPU Relationships

- Initially, the CPU had to load in the dataset from an API, likely the reason the CPU temp peaks at the start of the experiment.
- However, interestingly, we can see higher spikes of utilization despite a negative linear behaviour of the GPU temperature.
- As well, we can notice some minor fluctuations in GPU temp caused by the CPU

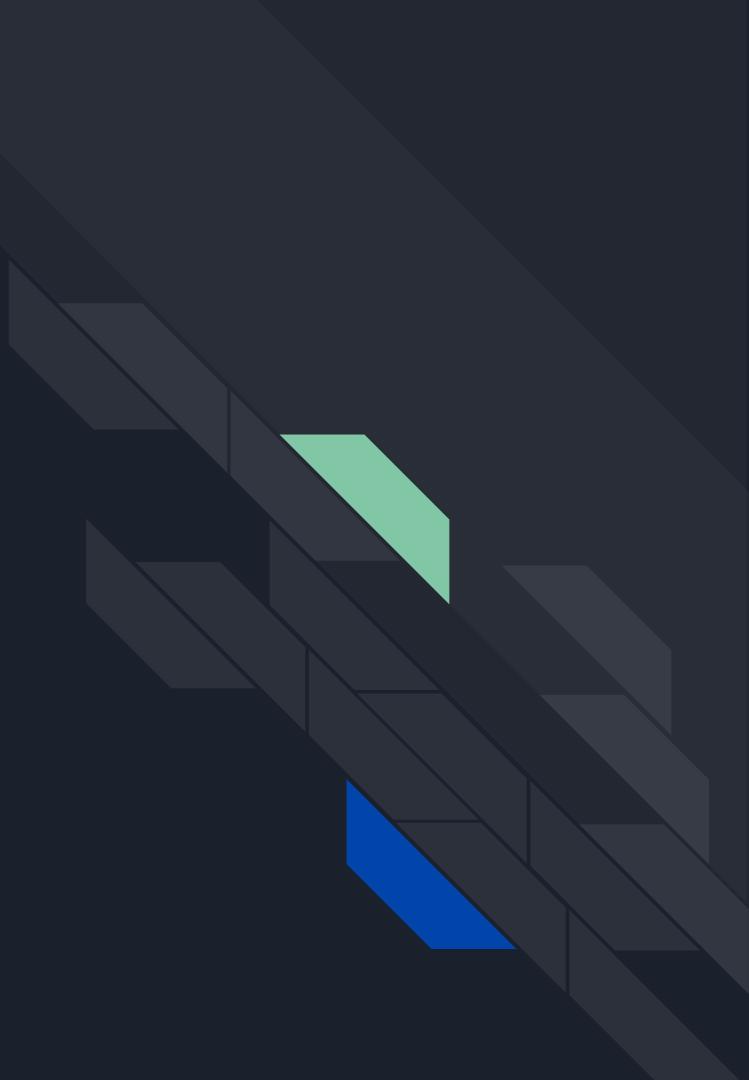


Feature Inter-Relationships

- Interestingly, with this lower GPU load, the correlations between the GPU and CPU is much more apparent.
 - This leads me to conclusion that the lower the GPU load, the more the CPU's behaviours has an affect on the GPU
- In addition, it seems that the CPU 1 average temperature and utilization has a stronger positive correlation with the GPU temperature than any of the other GPU diagnostic data.

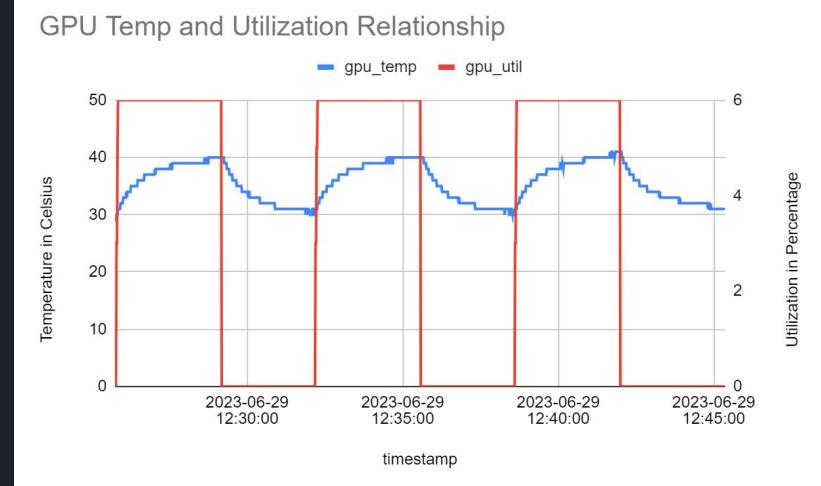
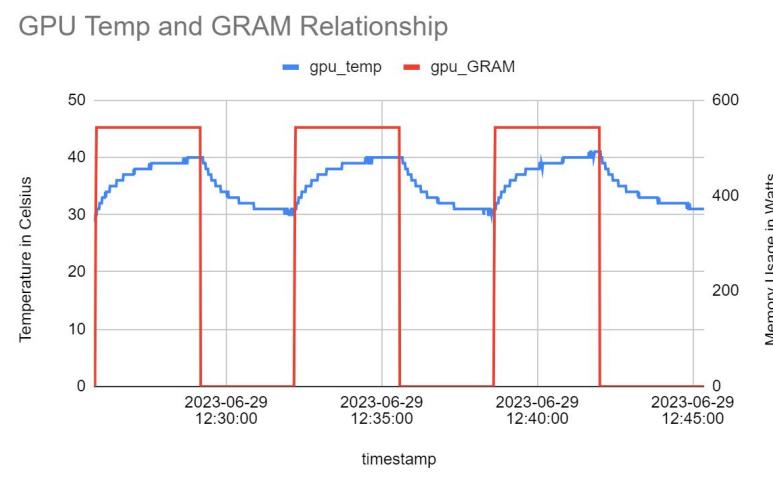


Sepia Filter using GPU



Sepia Filter using GPU

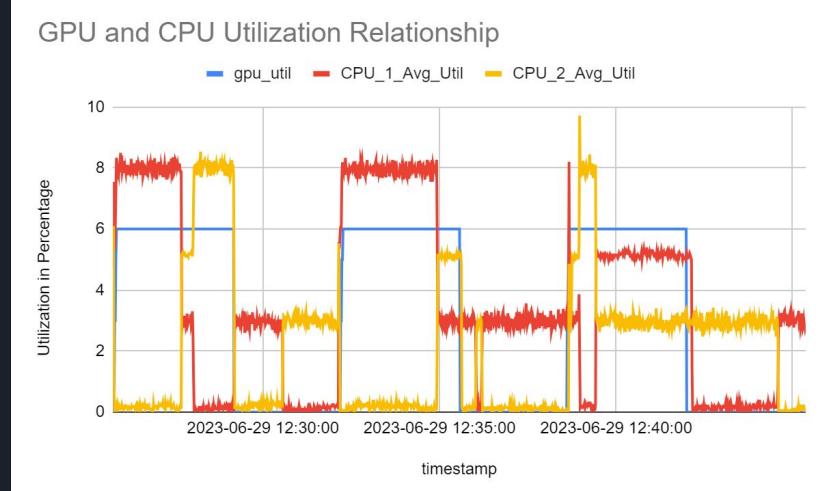
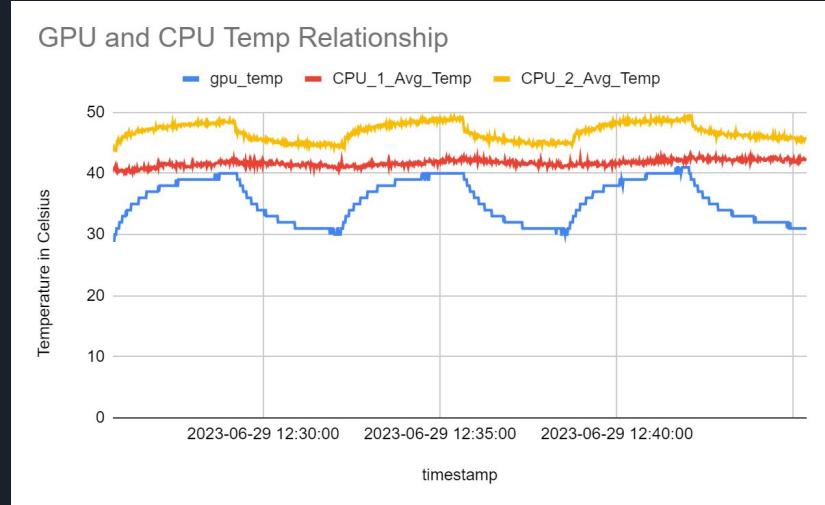
-



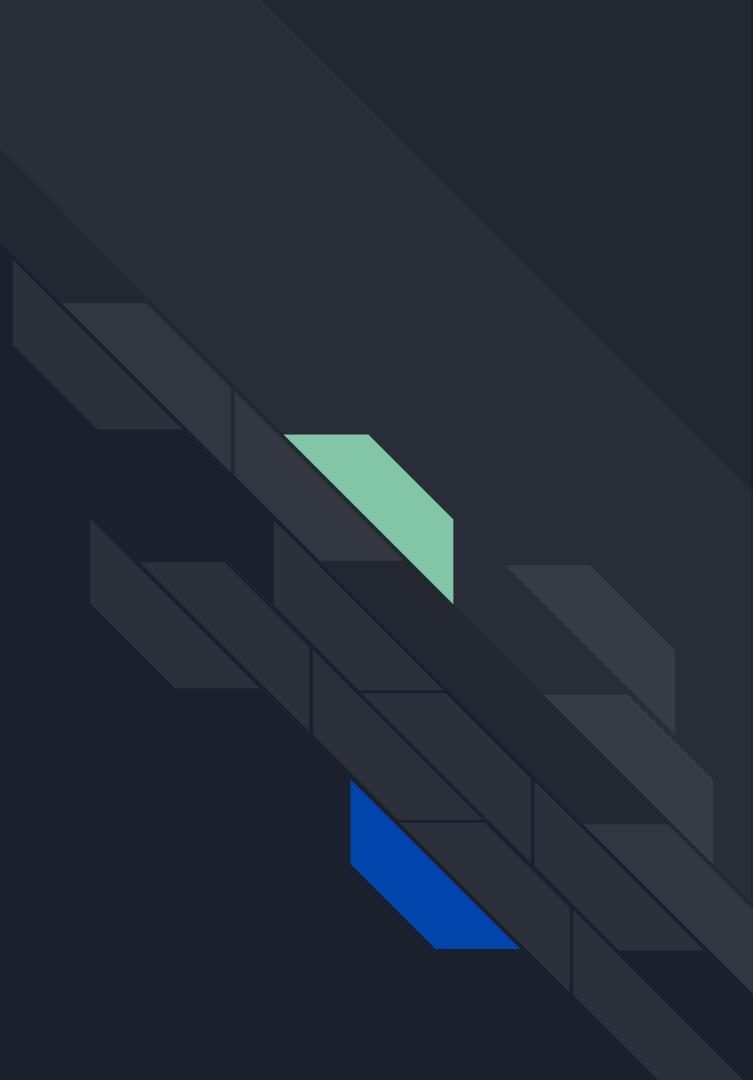
Sepia Filter using GPU

GPU and CPU Relationships

-



BlackScholes Model Using GPU



BlackScholes Model Using GPU

BlackScholes is a partial differential equation used for stock option pricing

- BlackScholes is extremely computationally expensive
 - Used extensively in Mathematical Finance
- Tries to determine fair prices for stock shares and contracts
- Used every day in the finance world
- Used to simulate real-world use case of GPU in the finance industry

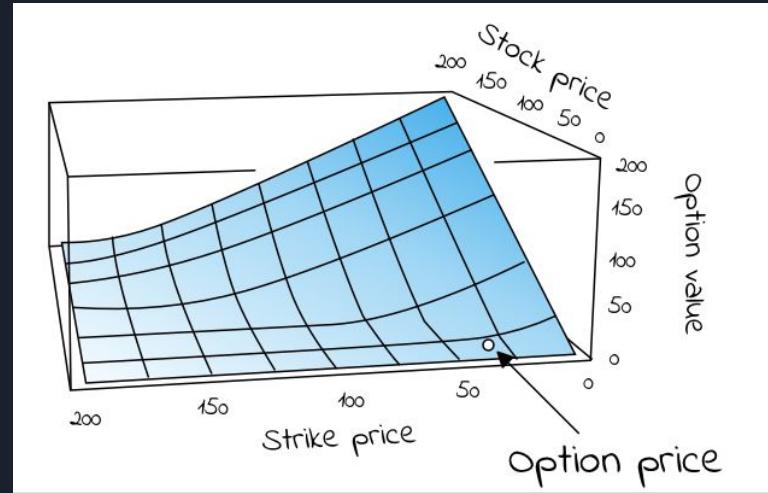


Figure Shows a visualization of BackScholes

Source:

<https://www.cantorsparadise.com/the-black-scholes-formula-explained-9e05b7865d8a>

```

import torch
from tqdm import tqdm as tqdm

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

def blackScholes_pyTorch(S_0, strike, time_to_expiry, implied_vol, riskfree_rate):
    S = S_0
    K = strike
    dt = time_to_expiry
    sigma = implied_vol
    r = riskfree_rate
    Phi = torch.distributions.Normal(0,1).cdf
    d_1 = (torch.log(S_0 / K) + (r+sigma**2/2)*dt) / (sigma*torch.sqrt(dt))
    d_2 = d_1 - sigma*torch.sqrt(dt)
    return S*Phi(d_1) - K*torch.exp(-r*dt)*Phi(d_2)

for i in tqdm(range(50000)):
    # Define your tensors and move them to the appropriate device right away
    S_0 = torch.tensor([100.], requires_grad = True, device=device)
    K = torch.tensor([101.], requires_grad = True, device=device)
    T = torch.tensor([1.], requires_grad = True, device=device)
    sigma = torch.tensor([0.3], requires_grad = True, device=device)
    r = torch.tensor([0.01], requires_grad = True, device=device)
    npv_pytorch = blackScholes_pyTorch(S_0, K, T, sigma, r)

```

The Black-Scholes *equation* is the partial differential equation (PDE) that governs the price evolution of European stock options in financial markets operating according to the dynamics of the Black-Scholes (sometimes Black-Scholes-Merton) model. The equation is:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS \frac{\partial V}{\partial S} - rV = 0$$

Equation 1. The Black-Scholes partial differential equation describing the price of a European call or put option over time

Figure Shows the BlackScholes Equation

Source:

https://medium.com/@quant_views/insight-into-black-scholes-equation-c9475c74cf53

BlackScholes Model Using GPU

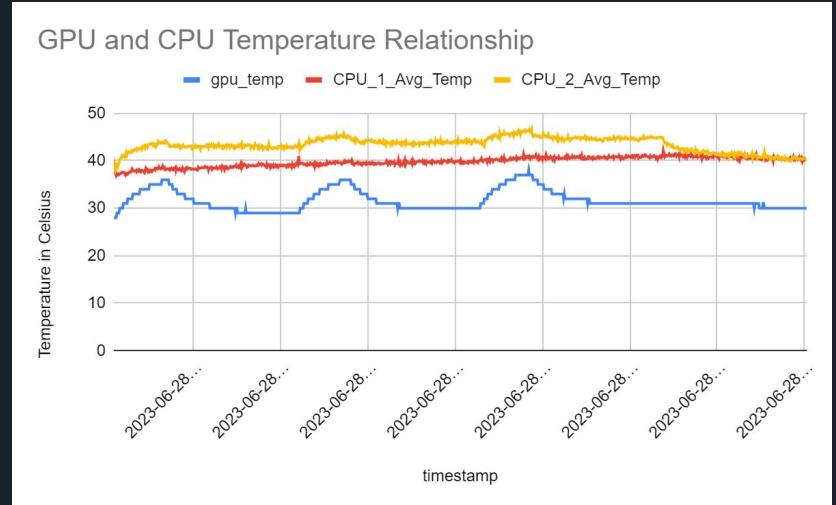
- The BlackScholes Model is a relatively low-intensity model despite its computational complexity.
 - The issue arises in Enterprise with the scale of the datasets
- With peaks of 8% GPU util, 420 MiB of GRAM, and 40 Degree Celsius, this experiment provides us with a great example of a low intensity workload



BlackScholes Model Using GPU

GPU and CPU Relationships

- Interestingly, there was a lot of activity in the CPU for this experiment
 - probably due to the way the code has been written, with some preliminary data generation on the CPU before executing the task on GPU
- Provides a nice real-world facsimile for balanced workloads where the CPU has a lot of affect on GPU

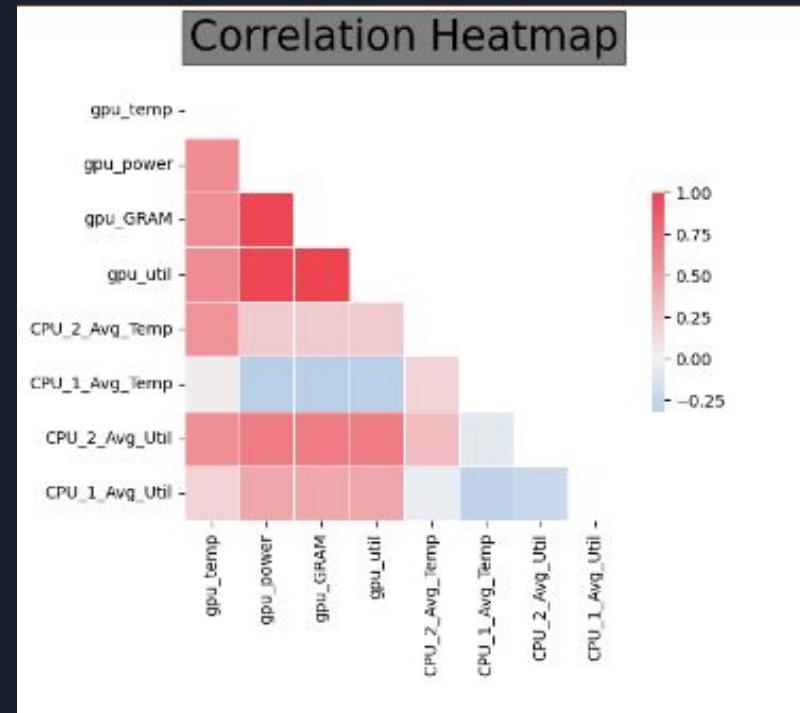


BlackScholes Model Using GPU

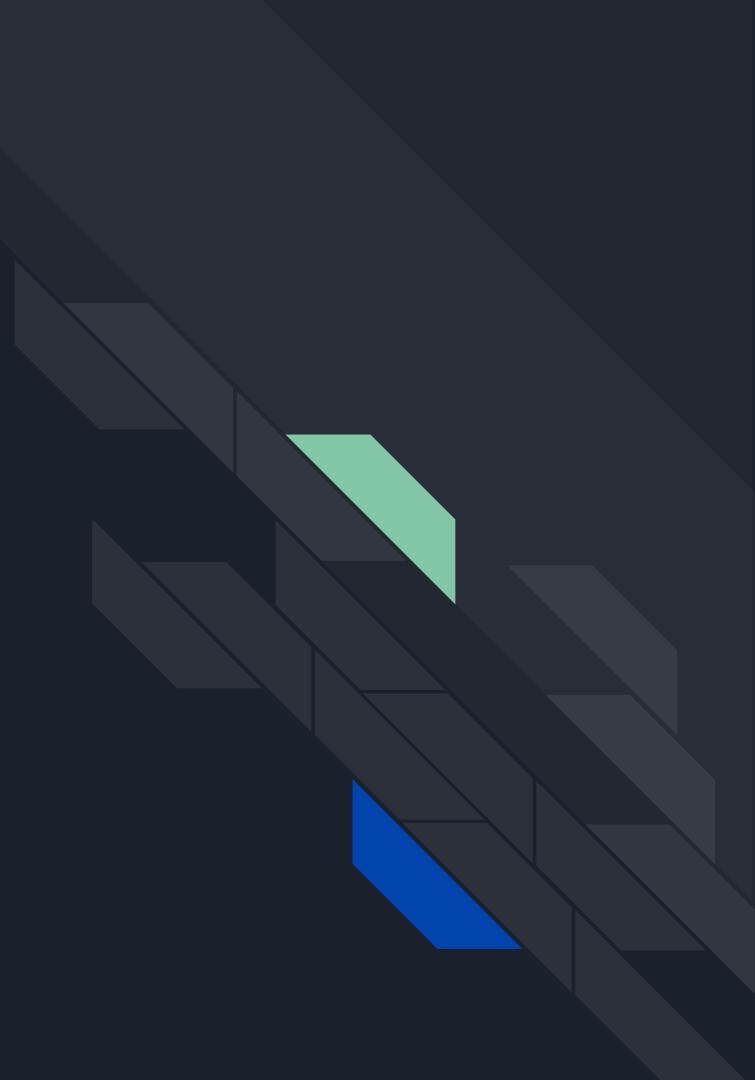


Feature Inter-Relationships

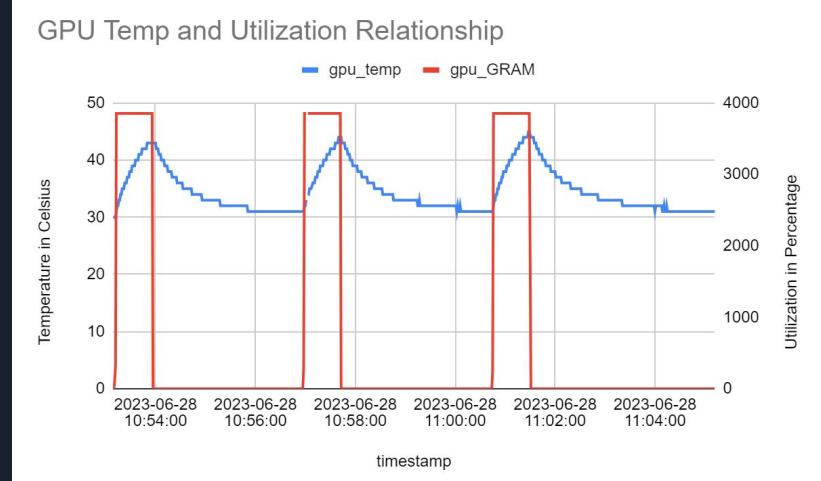
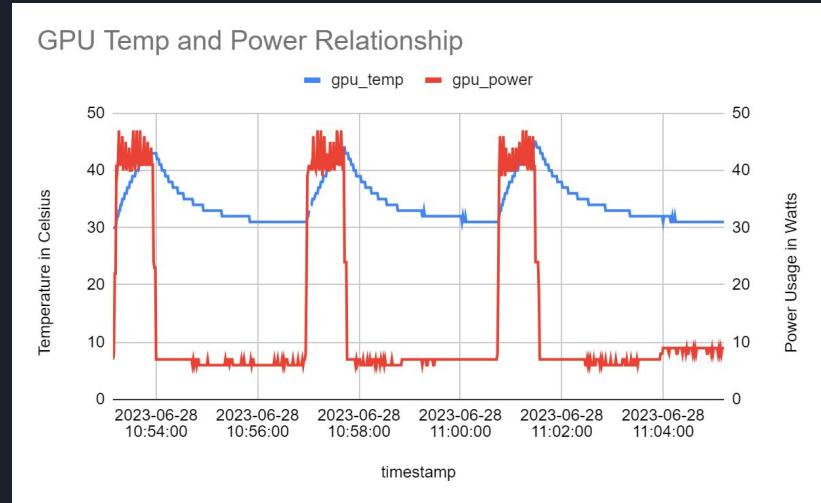
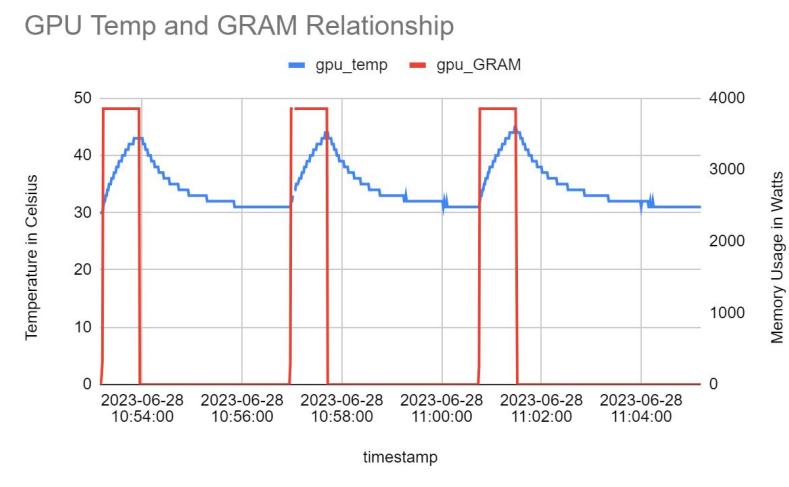
- Despite being a relatively similar workload in terms of temperature and utilization readings to the image processing experiment, there is no correlation with CPU 1 temperature and very low with the CPU 1 utilization
 - Furthermore, the rest of the features seem to share the same level of correlation to each other with the GPU Temp



MonteCarlo Pricing Options on GPU



MonteCarlo Pricing Options

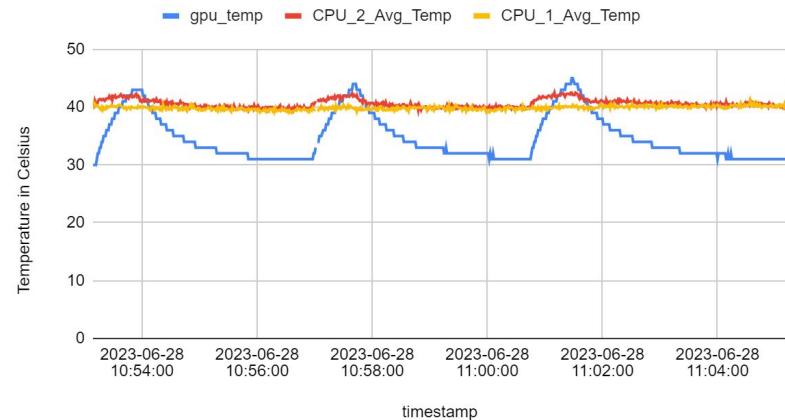


MonteCarlo Pricing Options

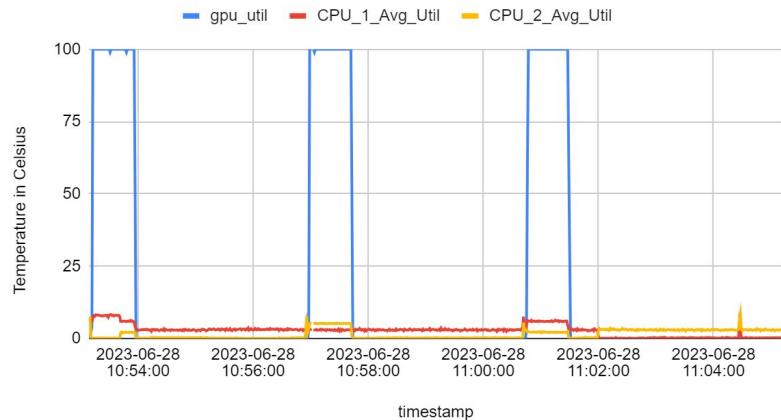
GPU and CPU Relationships

-

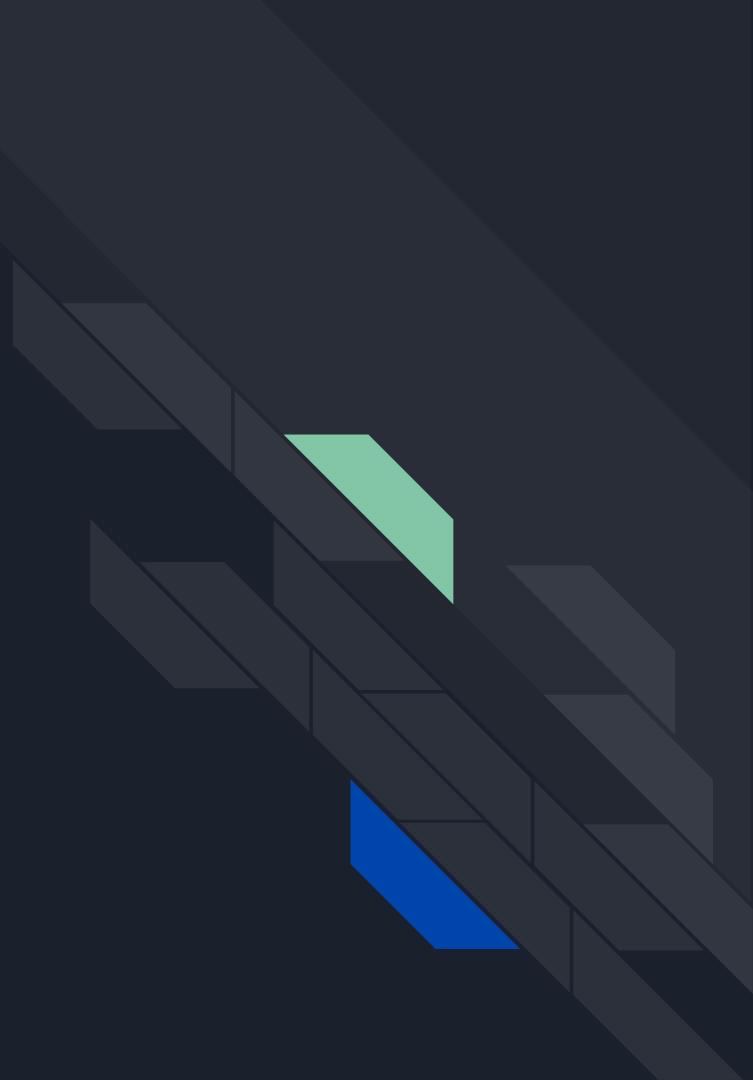
GPU and CPU Temp Relationship



GPU and CPU Utilization Relationship



Calculating Euclidean Distance using GPU



Calculating Euclidean Distance using GPU

Calculate Euclidean Distance between each respective point between two randomly generated Matrices

- Let one Matrix be X, and the other be Y.
- Process often used in recommendation systems, among other enterprise applications
- Parallelizable on GPU

$$\text{Euclidean Distance} = |X - Y| = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

X: Array or vector X

Y: Array or vector Y

x_i: Values of horizontal axis in the coordinate plane

y_i: Values of vertical axis in the coordinate plane

n: Number of observations



Figure Shows the formula for Euclidean Distance across 2 matrices

Source:

<https://www.geeksforgeeks.org/how-to-calculate-euclidean-distance-in-excel/>

```
import torch
from tqdm import tqdm as tqdm

# Define matrix size
N = 10000

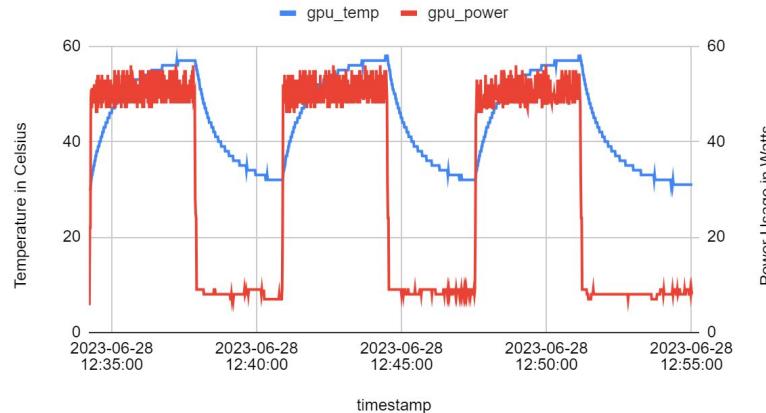
for i in tqdm(range(10000)):
    # Create two matrices on the GPU
    A = torch.rand(N, N, device='cuda').float()
    B = torch.rand(N, N, device='cuda').float()

    # Calculate the Euclidean distance between each pair of corresponding rows
    torch.sqrt(((A - B)**2).sum(dim=1))
```

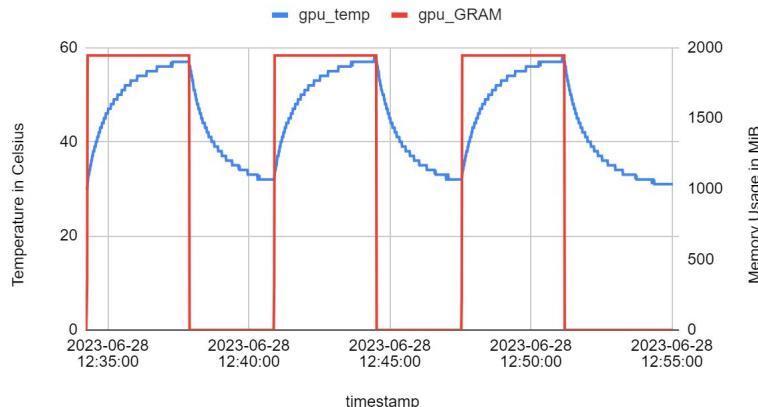
Calculating Euclidean Distance using GPU

- This benchmark has no visible anomalies in the data.
- The task follows a general medium-high pattern that has been emerging from our benchmarks.
- Even though the task is at 100% utilization, the memory usage and power consumption remains below what we've considered as high-intensity workloads

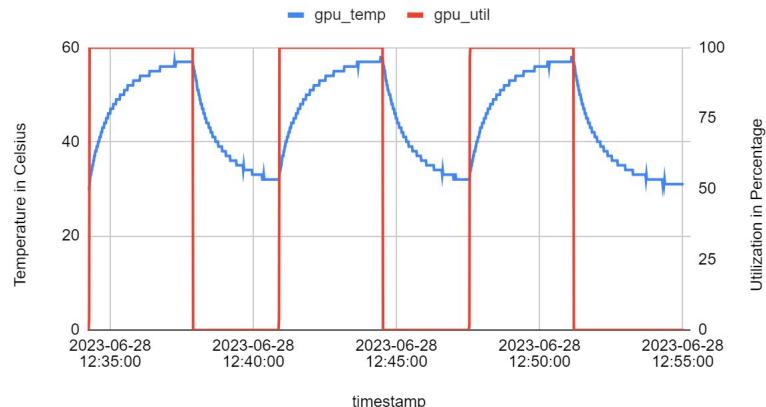
GPU Temp and Power Relationship



GPU Temp and GRAM Relationship



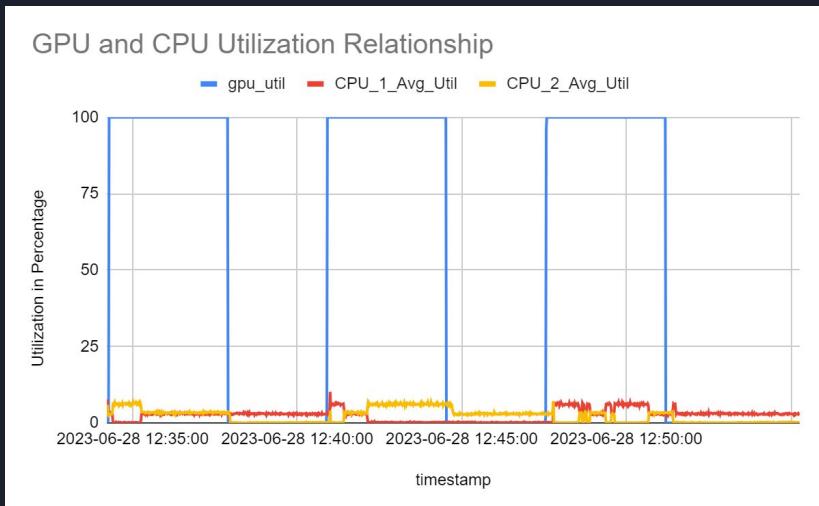
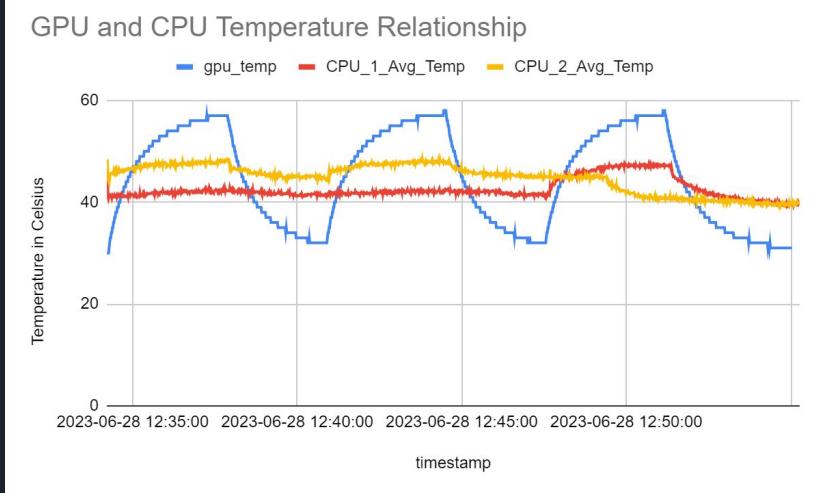
GPU Temp and Utilization Relationship



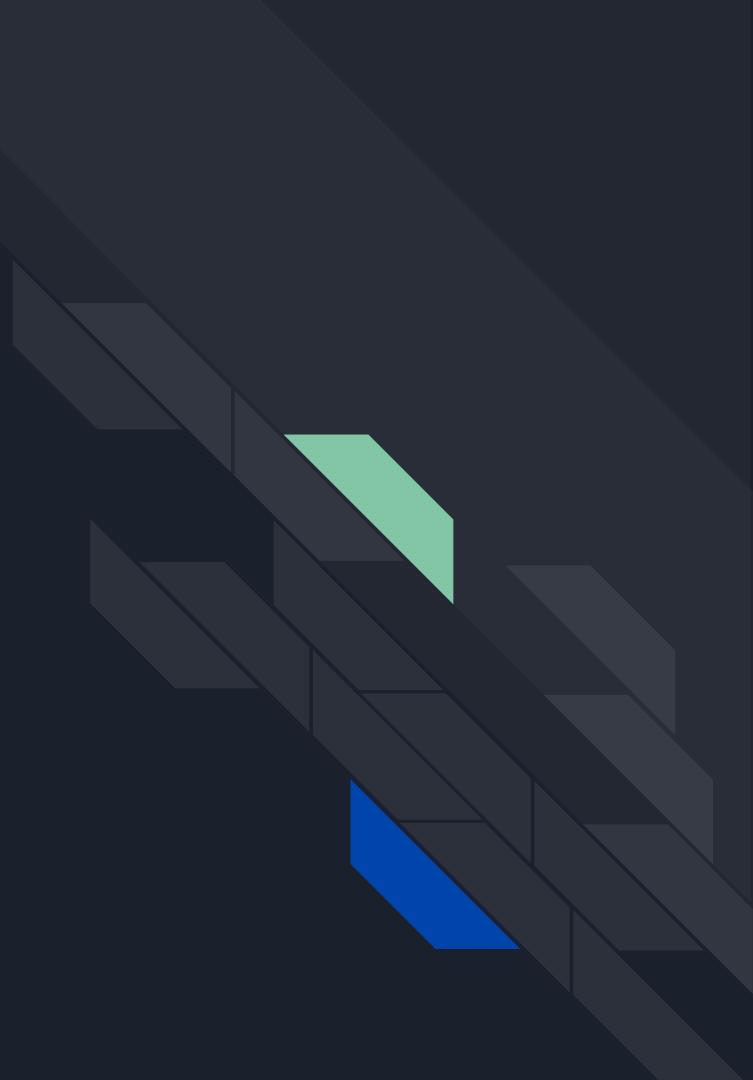
Calculating Euclidean Distance using GPU

GPU and CPU Relationships

- While there is some small correlation between the CPU and GPU for our diagnostic data, it all falls within the standard deviation of our experiments thus far



K-Means Clustering using GPU



K-Means Clustering using GPU

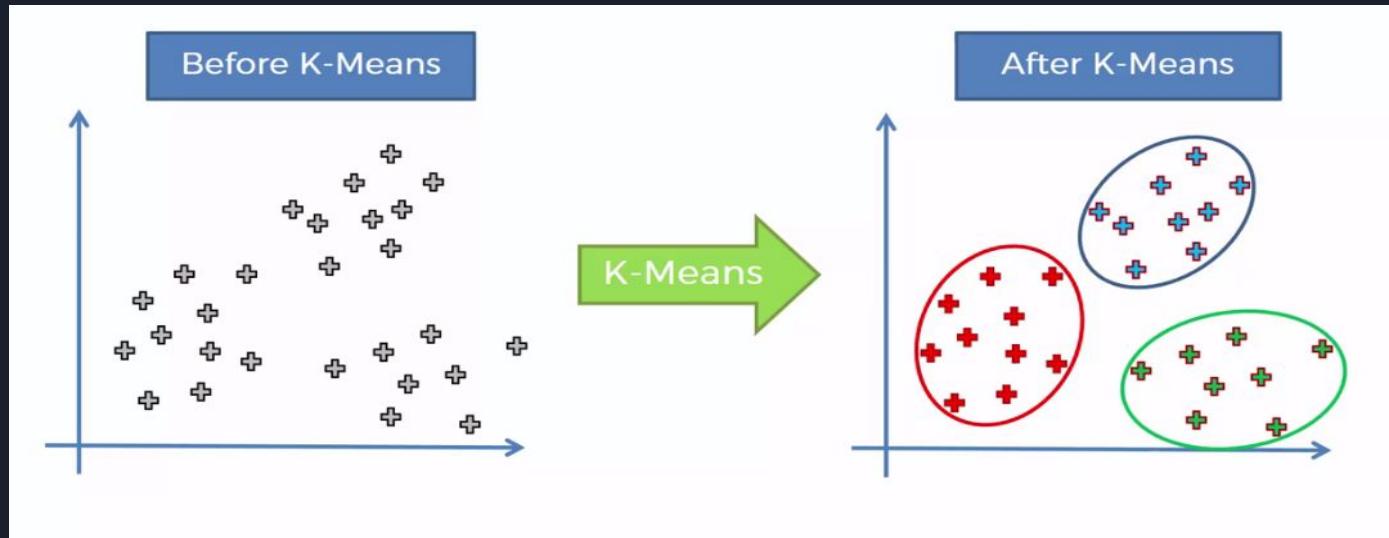
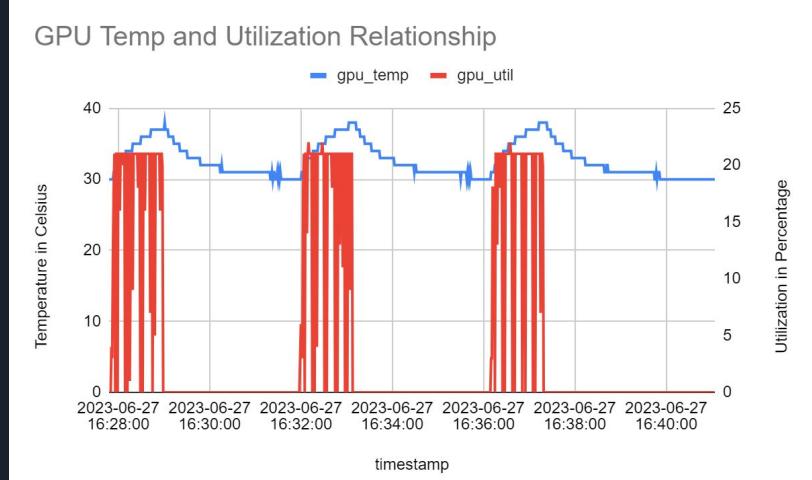
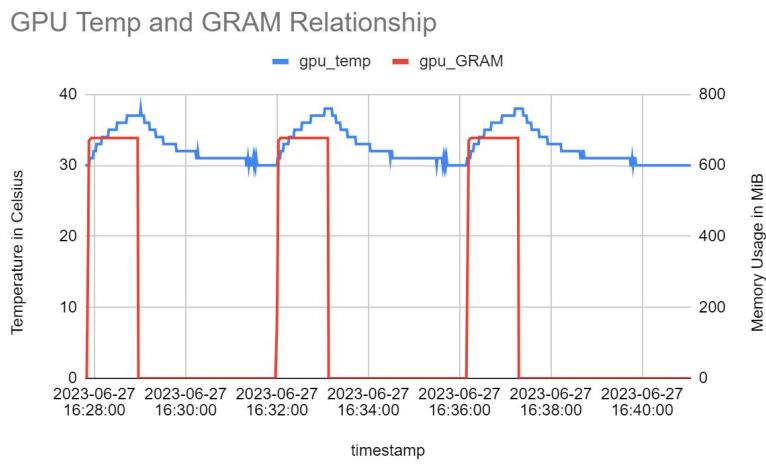


Figure Shows a visualization of k-means clustering

Source: <https://medium.datadriveninvestor.com/k-means-clustering-4a700d4a4720>

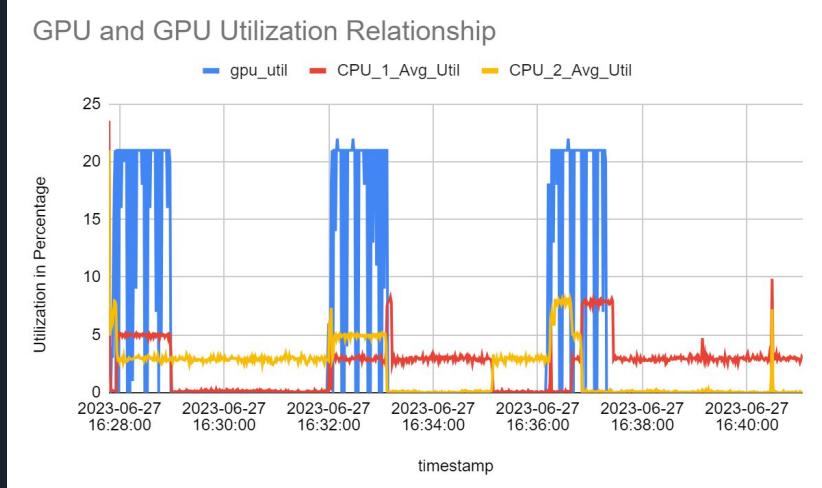
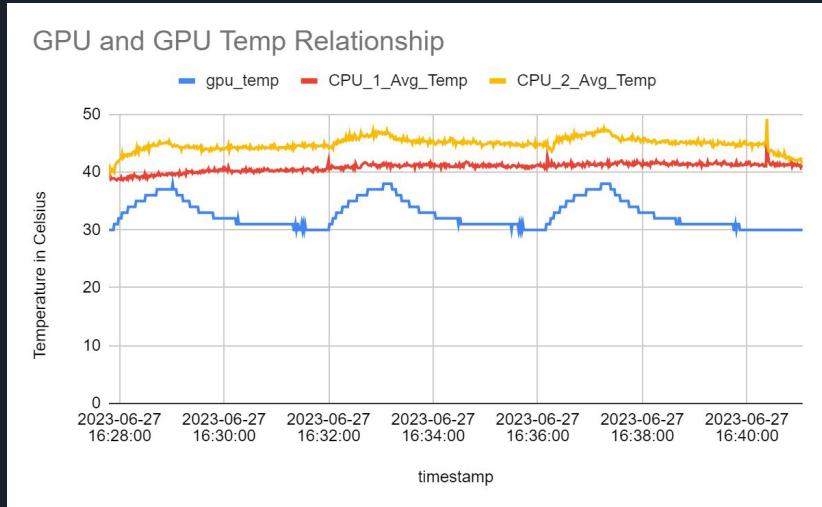
K-Means Clustering using GPU



K-Means Clustering using GPU

GPU and CPU Relationships

-



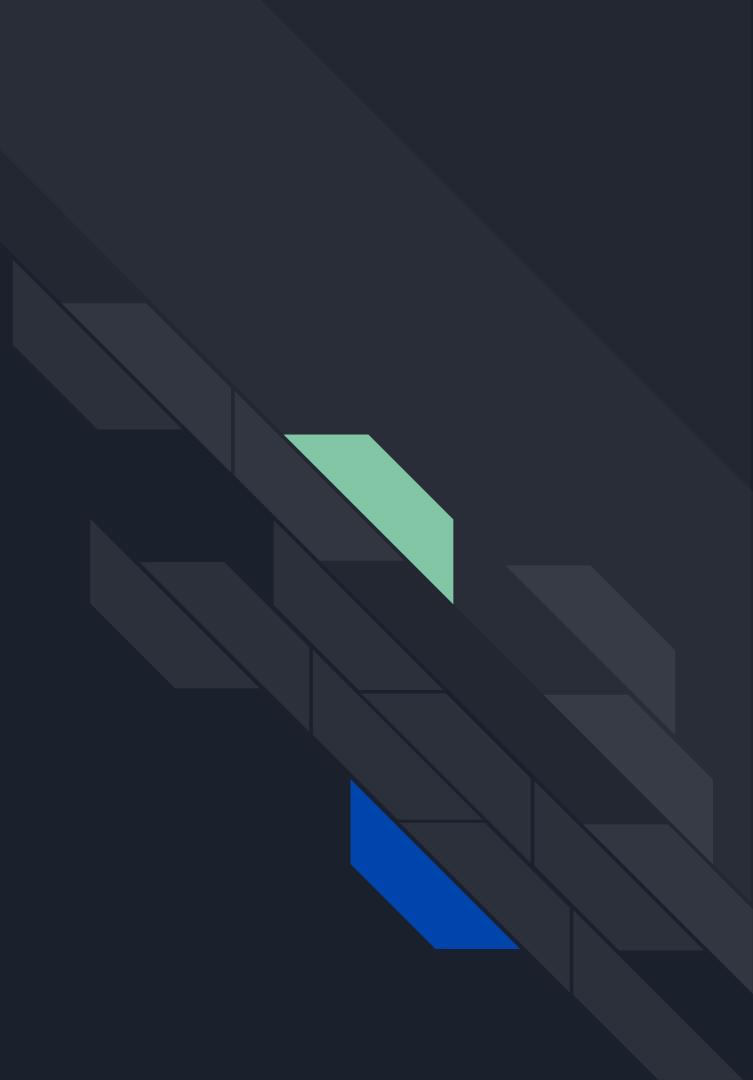
K-Means Clustering using GPU



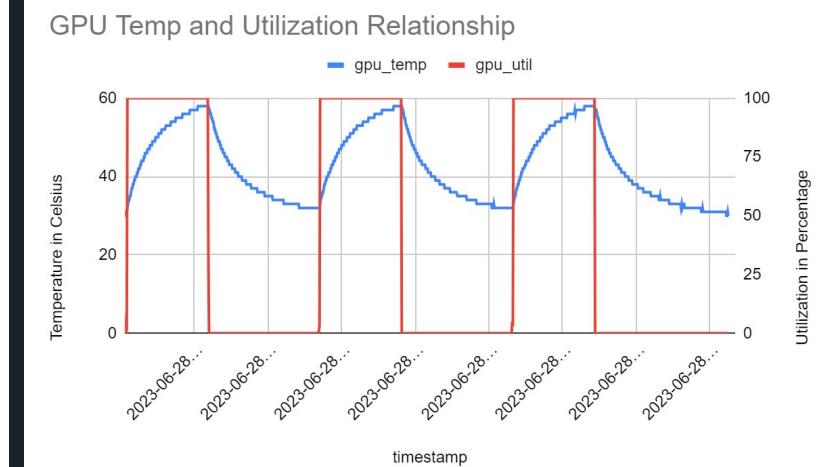
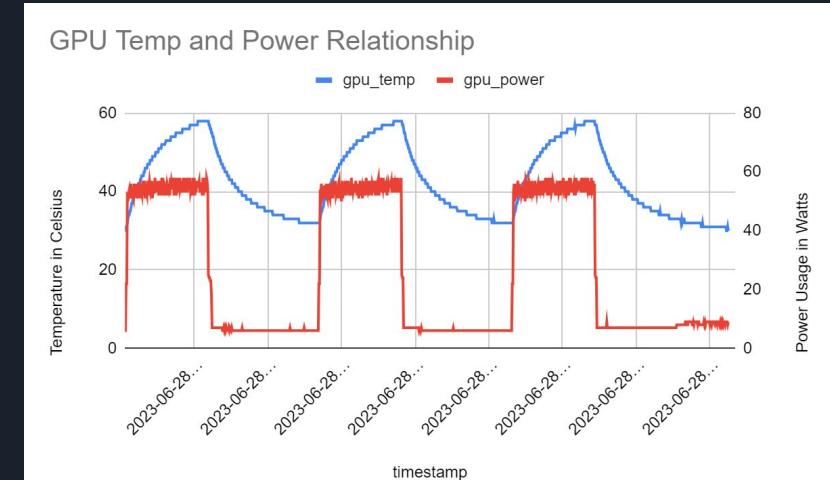
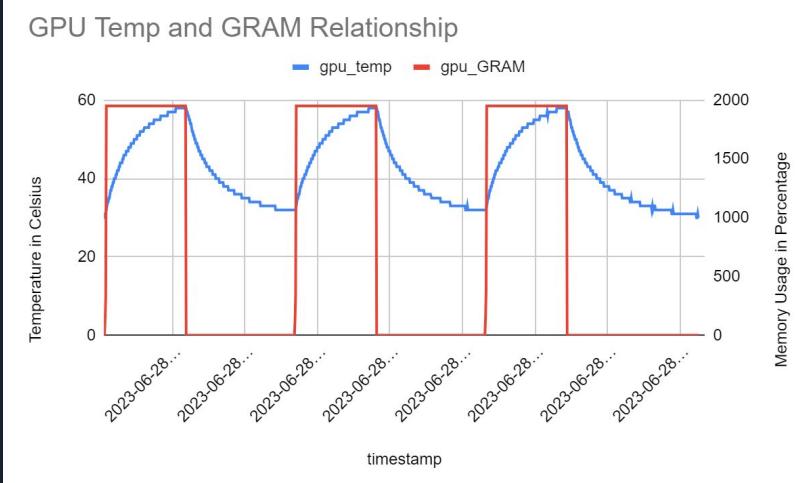
Feature
Inter-Relationships

-

Fast Fourier Transformations using GPU

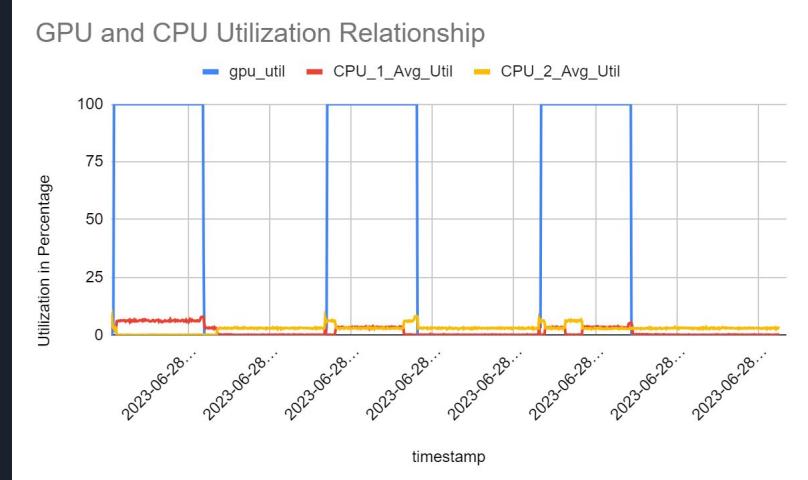
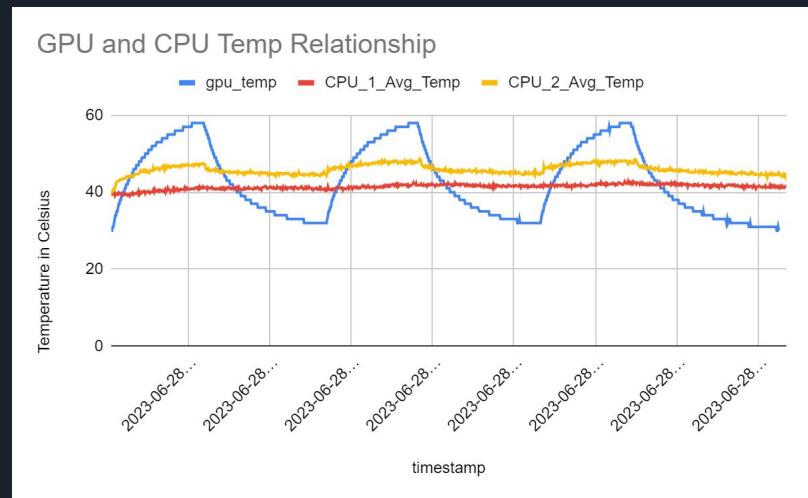


Fast Fourier Transformations using GPU

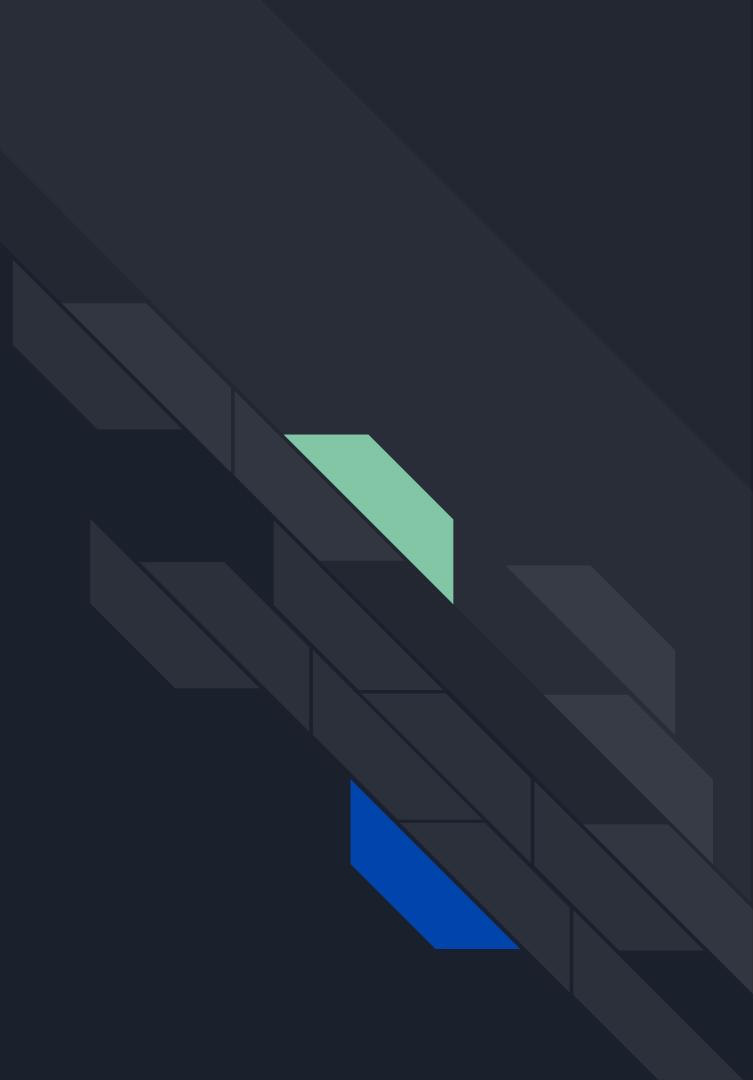


Fast Fourier Transformations using GPU

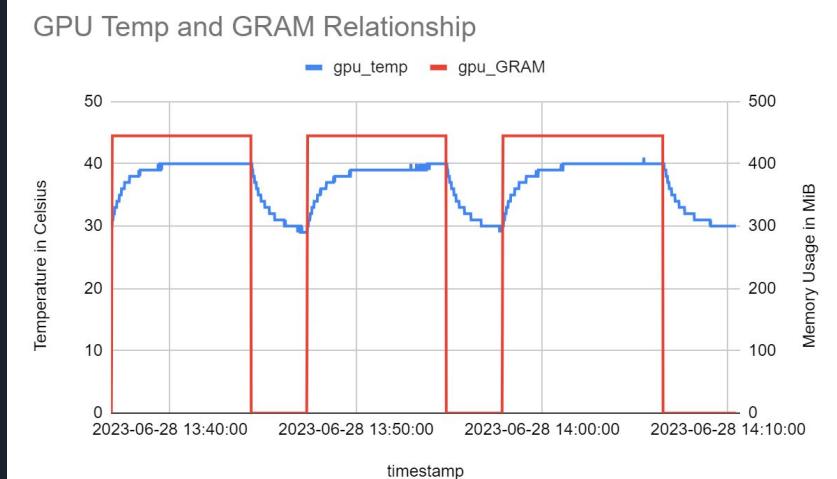
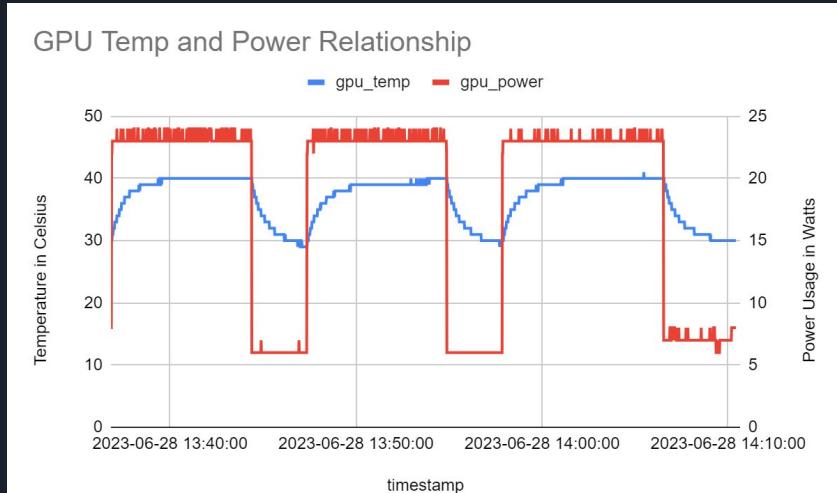
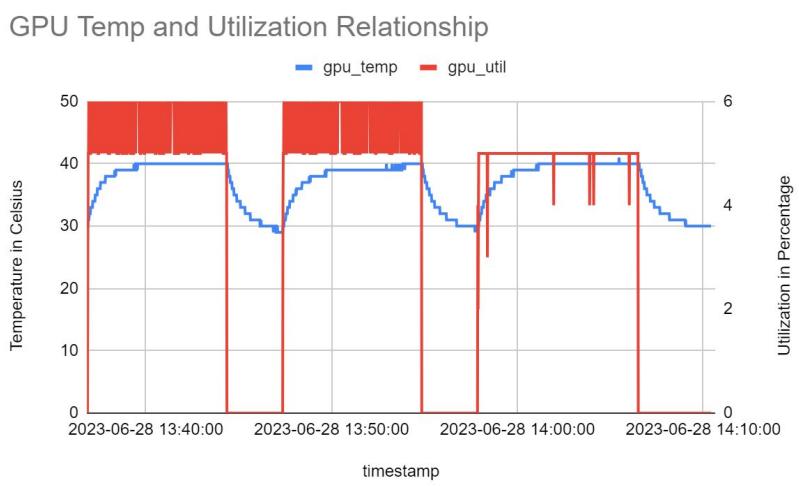
GPU and CPU Relationships



Spectrogram Transformations using GPU

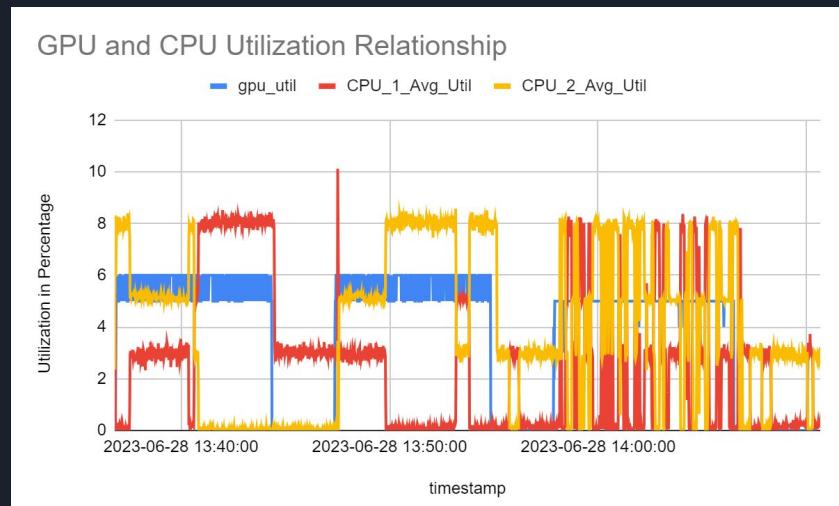
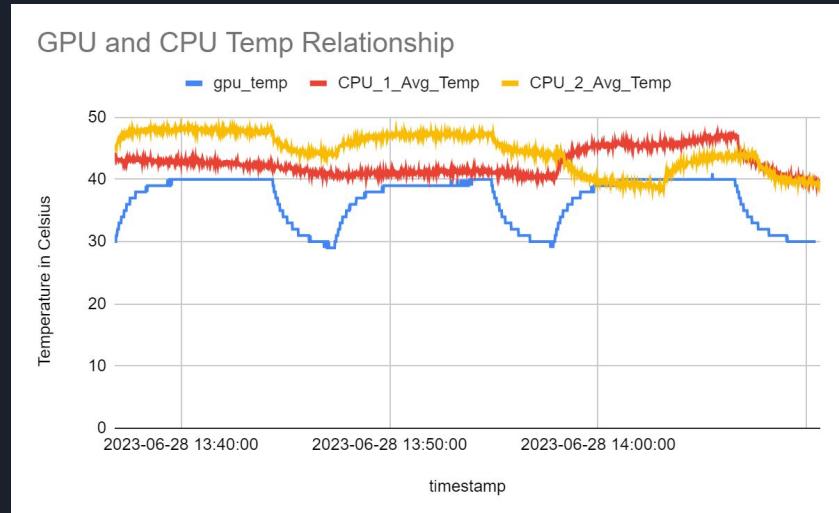


Spectrogram Transformations using GPU

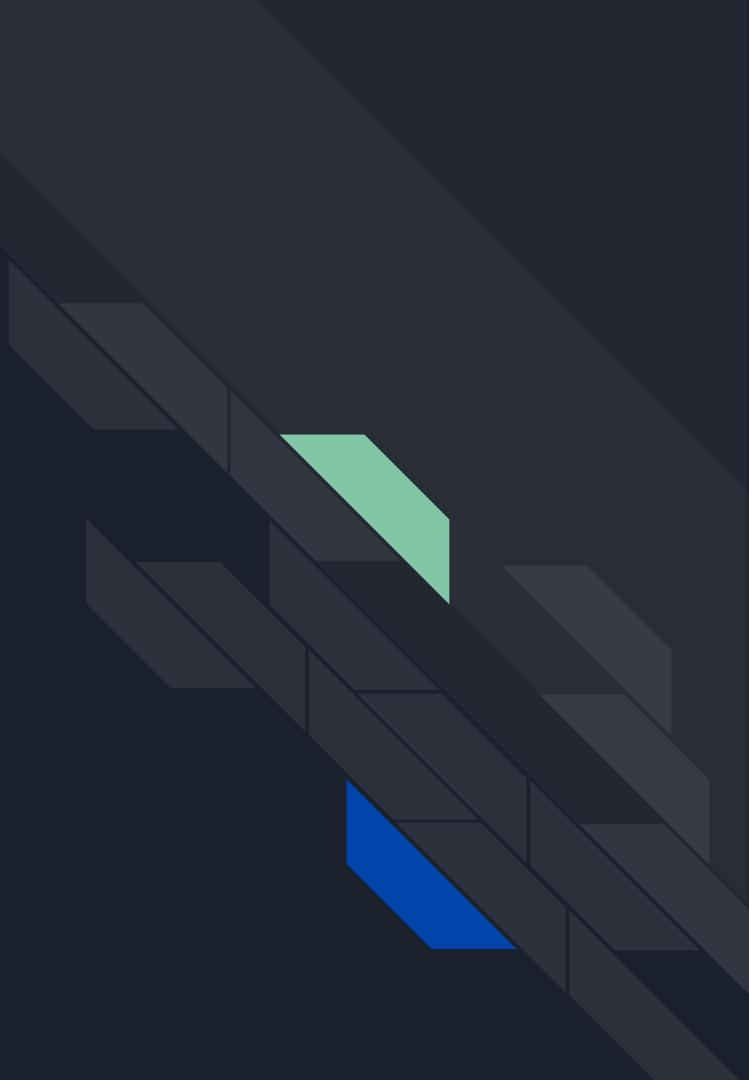


Spectrogram Transformations using GPU

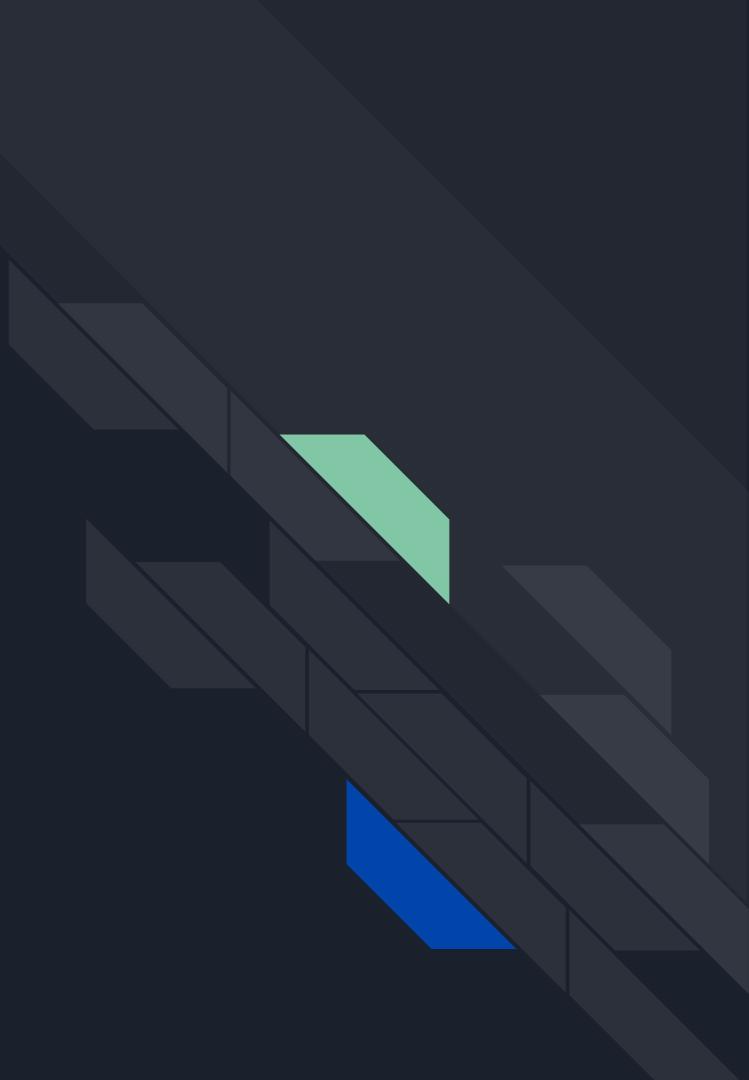
GPU and CPU Relationships



Preliminary Modelling



DistilBERT Data Only



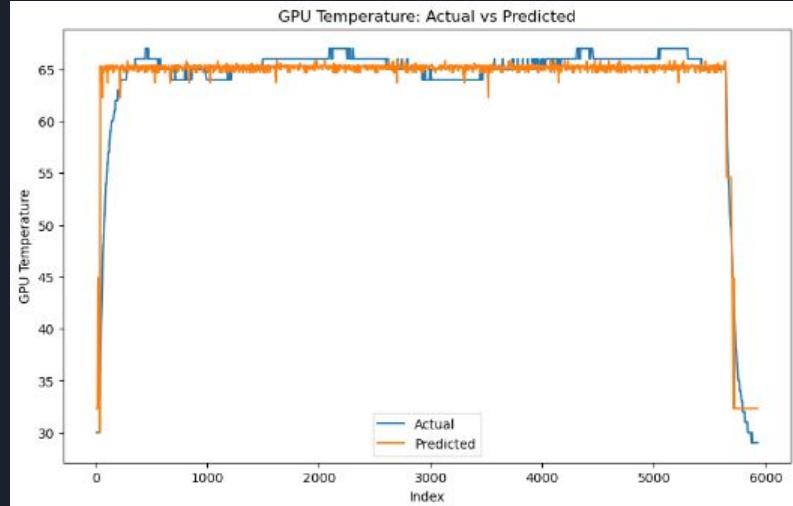
Linear Regression with XGBoost

- Linear, bi-variate predictions do not provide enough insight for the model to provide accurate predictions, as the number of correlative factors that have an impact on temp readings are not being included

Linear Regression

```
X = df['gpu_power']
y = df['gpu_temp']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, random_state=42)
model = xgb.XGBRegressor()
model.fit(X_train, y_train)
preds = model.predict(X_test)
rmse = np.sqrt(mean_squared_error(y_test, preds))
print(f'RMSE: {rmse}')

RMSE: 2.6366004622676544
```



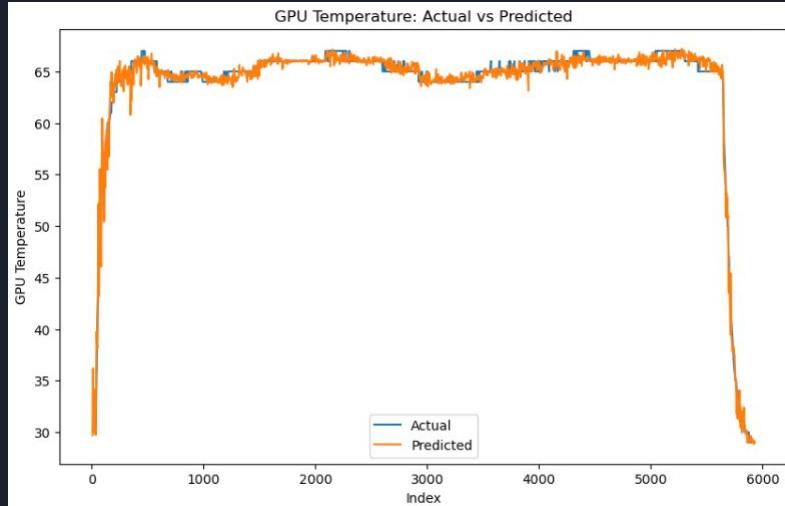
Polynomial Regression with XGBoost

- Including all of our features and performing a polynomial regression is performing much better “out of the box”, although its performance is relatively lacking at the moment
 - It is a really good baseline and “jumping off point” with which to probe further modelling

Polynomial Regression

```
X = df.drop(columns = ['gpu_temp', 'timestamp'])
y = df['gpu_temp']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, random_state=42)
model = xgb.XGBRegressor()
model.fit(X_train, y_train)
preds = model.predict(X_test)
rmse = np.sqrt(mean_squared_error(y_test, preds))
print(f'RMSE: {rmse}')
```

RMSE: 0.7967018531426376



LSTM using single variable

- Single variate time series prediction
- Although not a directly “apples-to-apples” comparison, on the test set, the LSTM is well out-performing XGBoost “out-of-the-box”
- Would be worthwhile to implement some kind of multivariate LSTM model
 - This should push performance close to .1 RMSE score, which is the performance needed to inform a load-balancing algorithm

```
# Make predictions
test_predictions = model_best.predict(X_test).flatten() # Without flatten, its that same double

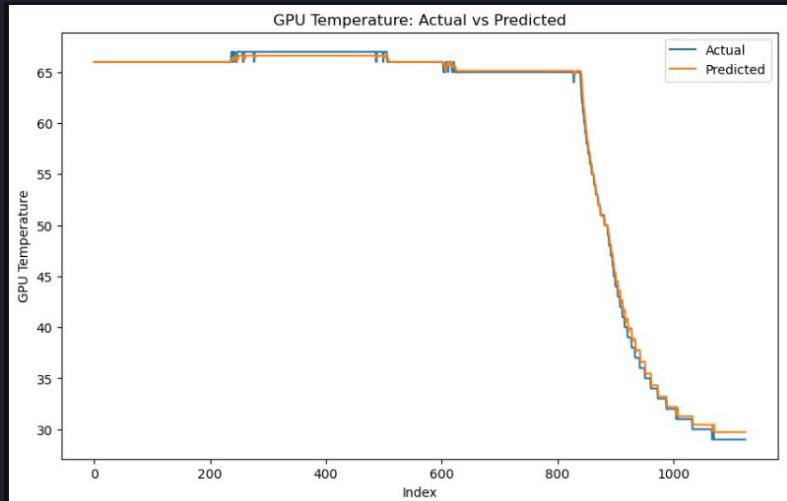
# Return results of best model as Dataframe
results = pd.DataFrame(data = {"Test Predictions": test_predictions, 'Actuals': y_test})
results = results[:1125]

39/39 [=====] - 0s 2ms/step

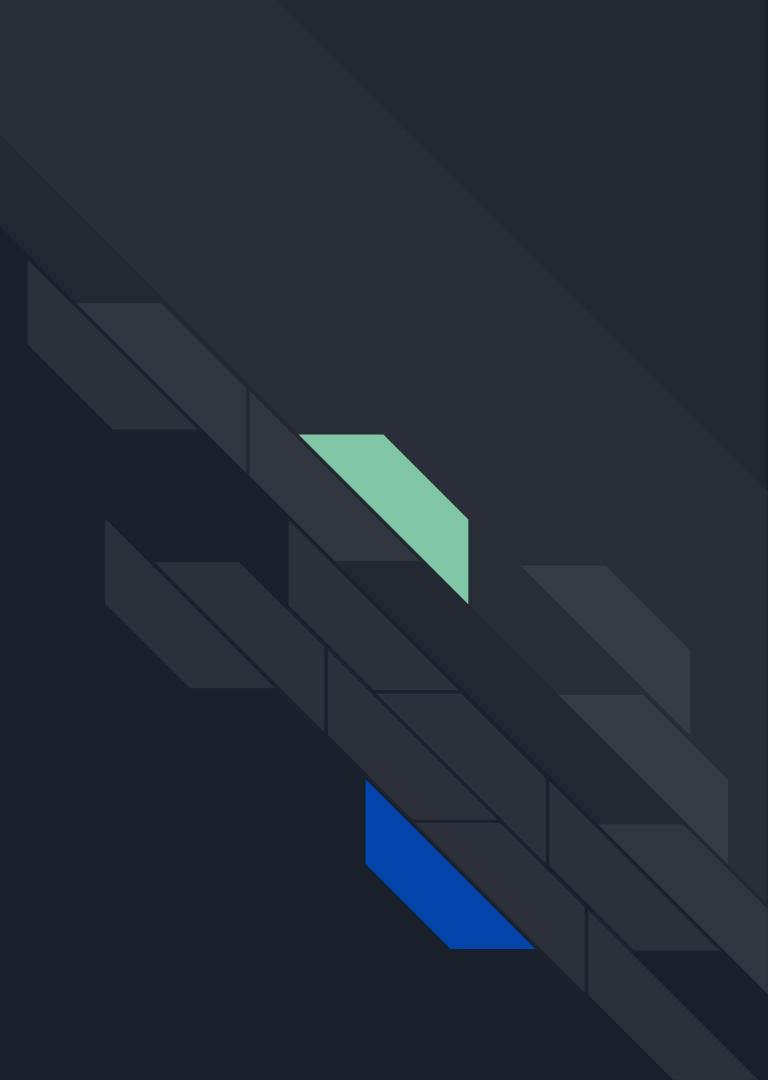
from sklearn.metrics import mean_squared_error

# Calculate and display RMSE for best model
rmse = np.sqrt(mean_squared_error(results['Actuals'], results['Test Predictions']))
print(f'RMSE: {rmse}')

RMSE: 0.4155781219100762
```



Matmul, BERT, DistilBERT,
Image Classifier, and
BlackScholes Data
Combined



Linear Regression with XGBoost

Linear Regression

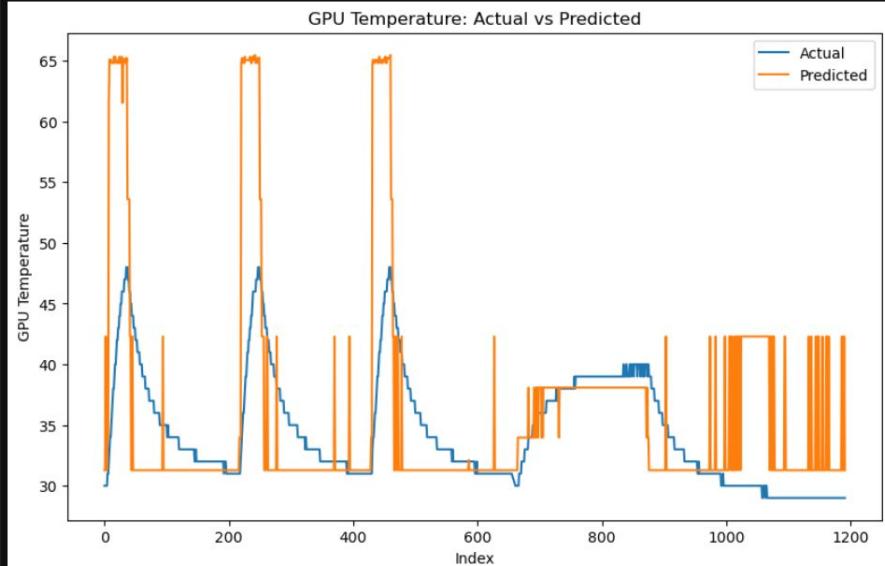
```
X_train = train_df['gpu_power']
y_train = train_df['gpu_temp']

X_test = test_df['gpu_power']
y_test = test_df['gpu_temp']

model = xgb.XGBRegressor()
model.fit(X_train, y_train)
preds = model.predict(X_test)
rmse = np.sqrt(mean_squared_error(y_test, preds))
print(f'RMSE: {rmse}')

RMSE: 7.917304238322233
```

- Right away with our more varied dataset, we can see that any kind of bivariate, linear model will be completely insufficient to extrapolate insights from our data
 - With so much variation, and such a simple model, this outcome is to be expected, however it proves to us an important point of the in-feasibility of using a bivariate XGBoost to inform our scheduling algorithms



Polynomial Regression with XGBoost

- Despite being a polynomial model using all of our features, XGBoost isn't built to find patterns in such varied data, which is much more akin to real world situations.
 - This presents to us a serious concern of using this model for anything other than declaring a low-bar performance benchmark

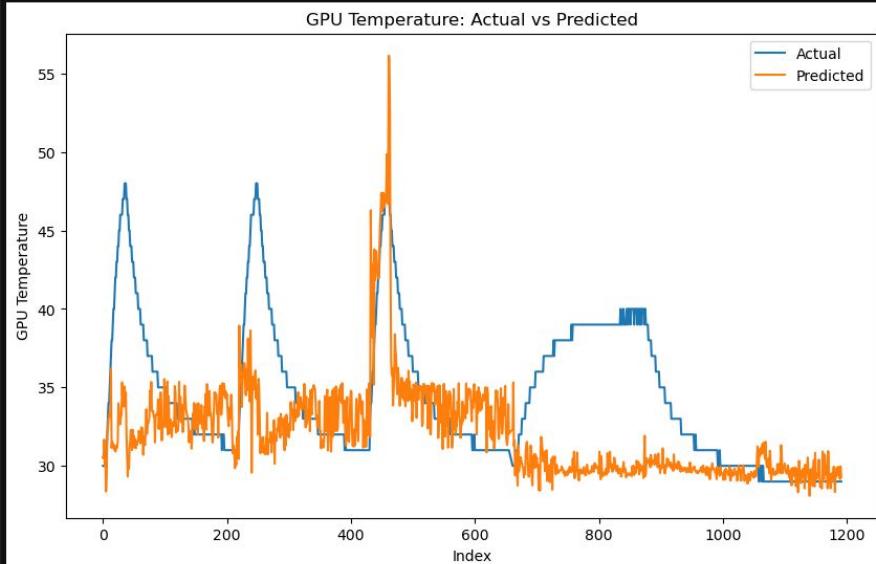
Polynomial Regression

```
X_train = train_df.drop(columns = ['gpu_temp', 'timestamp'])
y_train = train_df['gpu_temp']

X_test = test_df.drop(columns = ['gpu_temp', 'timestamp'])
y_test = test_df['gpu_temp']

model = xgb.XGBRegressor()
model.fit(X_train, y_train)
preds = model.predict(X_test)
rmse = np.sqrt(mean_squared_error(y_test, preds))
print(f'RMSE: {rmse}')

RMSE: 5.219650118166221
```



LSTM using single variable

- Although not performing fantastically, the LSTM model, despite using a single variable, is doing orders of magnitude better than the XGBoost model for regression.
- This points to the necessity of using some kind of recurrent neural network that can self correct through some kind of back-propagation algorithm to develop the necessary set of weights to match complex, irregular data to the level of accuracy necessary to create a generalizable solution to the problem we are trying to tackle with our research

```
# Make predictions
test_predictions = model_best.predict(X_test).flatten() # Without flatten, its that same double

# Return results of best model as Dataframe
results = pd.DataFrame(data = {"Test Predictions": test_predictions, 'Actuals': y_test})
results = results[:1125]

39/39 [=====] - 0s 2ms/step

from sklearn.metrics import mean_squared_error

# Calculate and display RMSE for best model
rmse = np.sqrt(mean_squared_error(results['Actuals'], results['Test Predictions']))
print(f'RMSE: {rmse}')

RMSE: 0.4155781219100762
```

