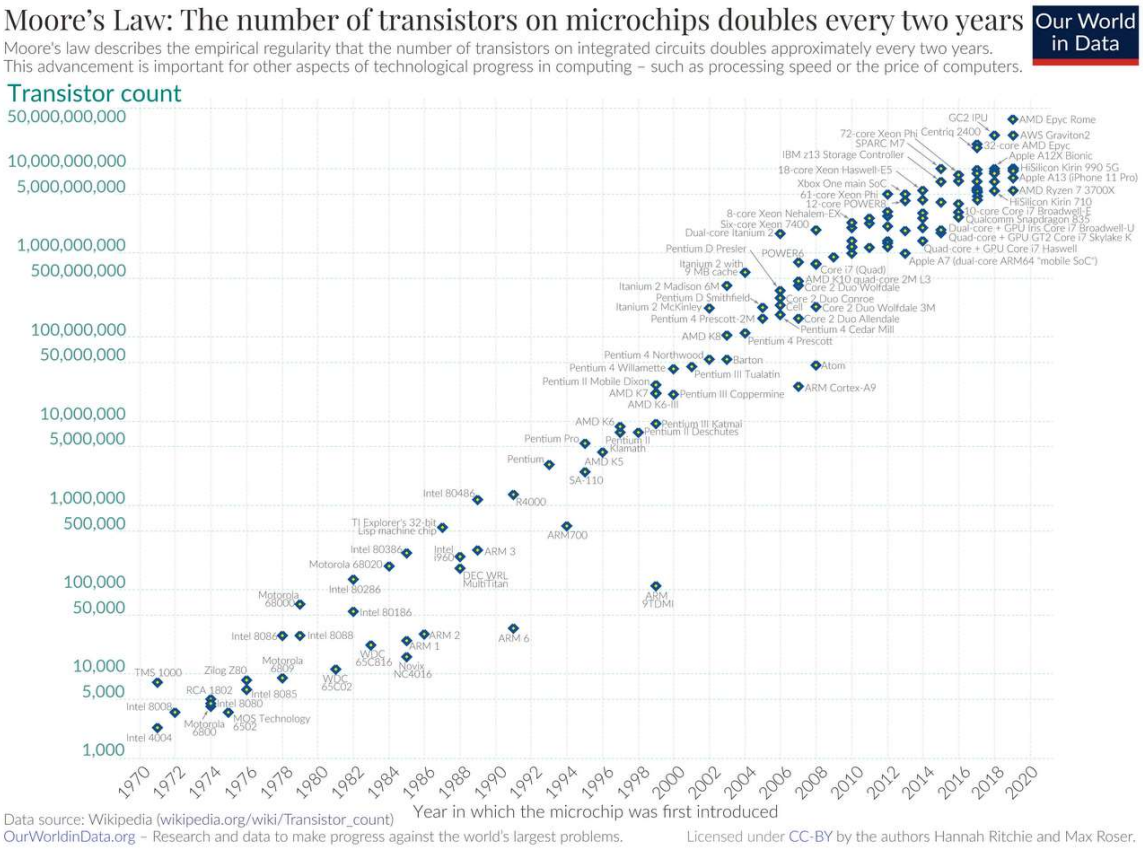
CSCA 5622 Final Project - Consumer PC Hardware Trends and Predictions

By Moshiur Howlader

Introduction

In the digital age, computer hardware is ubiquitous, and its performance continues to improve year by year. Intel's co-founder provided valuable insight into how computers would evolve, known as **Moore's Law** (see Wikipedia). This observation states that the number of transistors in an integrated circuit (IC) doubles approximately every two years. The chart below illustrates the trend from 1970 to 2020:



Based on Moore's Law, consumers might expect to get computer hardware with double the transistors every two years—leading to predictable and consistent increases in computing power. However, the reality is far more complex. As the number of transistors crammed into a fixed area increases, **quantum physics** begins to interfere, imposing physical limitations. These constraints prevent engineers from continuing to follow Moore's Law indefinitely.

According to nano.gov, the average size of a gold atom is 1/3 nm! Clearly, there is a limit to how many transistors can be packed into computer parts. Below are the trends in chip lithography size according to Wikipedia:

Feature Size	Year
20 µm	1968
10 µm	1971
6 µm	1974
3 µm	1977
1.5 µm	1981
1 µm	1984
800 nm	1987
600 nm	1990
350 nm	1993
250 nm	1996
180 nm	1999
130 nm	2001
90 nm	2003
65 nm	2005
45 nm	2007
32 nm	2009
28 nm	2010
22 nm	2012
14 nm	2014
10 nm	2016
7 nm	2018
5 nm	2020
3 nm	2022
2 nm	~2025 (Future)

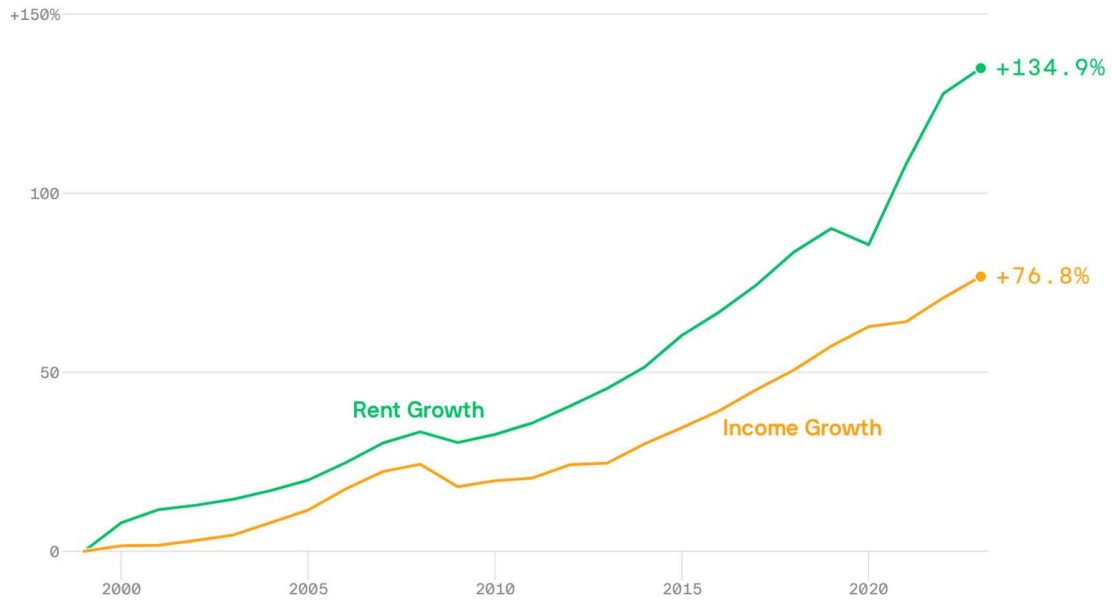
According to Jensen Huang, the CEO of Nvidia, **Moore's Law is dead** (TechSpot article). This statement seems reasonable given the physical limitations of current chip designs. As the rate of improvement in transistor count decreases year over year, will consumers start paying more for diminishing performance gains?

Why Should Consumers Care About the Death of Moore's Law?

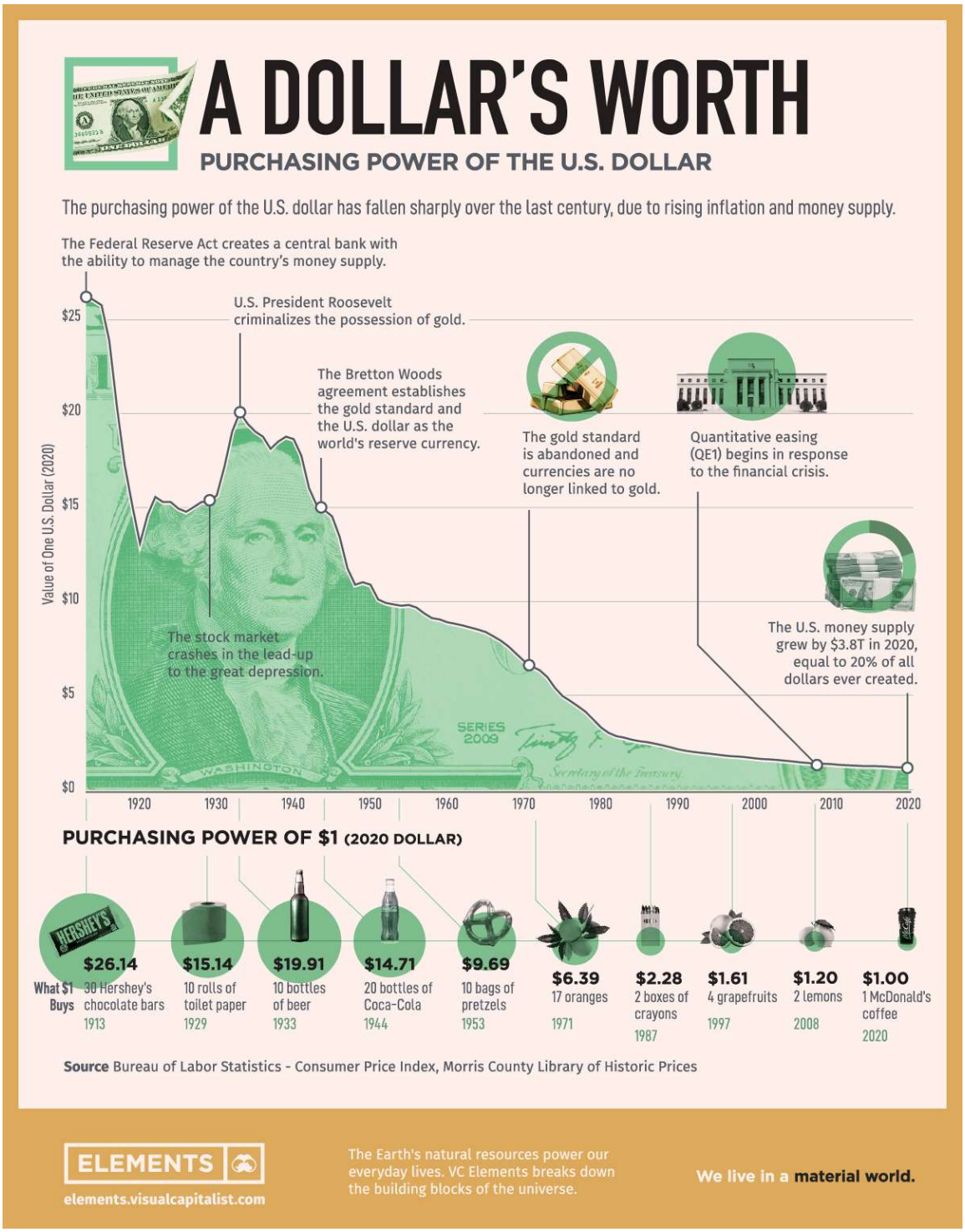
With the decline of Moore's Law, we can expect fewer improvements in transistor density in upcoming generations. This poses a concern for consumers, as we may start paying more for diminishing returns on performance. As traditional computing approaches its physical limits, incremental improvements will become smaller, potentially benefiting corporations more than consumers. This could lead to a scenario where consumers pay more for fewer benefits, which is undesirable.

The Economic Reality Today

Inflation has steadily eroded purchasing power in the USA over the last 50 years. As inflation rises, the real cost of consumer goods, including technology, increases, affecting affordability. Here are links to inflation-related data:



Source: Axios: America's Growing Rent Burden



Source: Visual Capitalist: Purchasing Power of the U.S. Dollar Over Time

Purpose of This Project

This project aims to answer the following key questions:

1. What are the trends in CPU and GPU parts over the past 20 years?

2. Is the price-to-performance ratio of these parts keeping up? Are consumers getting a fair deal compared to 10 to 20 years ago?
3. Can we predict the performance of next-gen, unreleased CPU and GPU parts using supervised machine learning models?

To achieve these goals, we focus on supervised learning tasks that involve predicting future performance and pricing. We use regression models like linear regression, random forest, XGBoost and Gradient Boosting to analyze trends and make predictions based on historical data.

Data Collection & Description

The data for both CPU/GPU was collected from via web scraping:
https://www.techpowerup.com/

Note that various other sources were considered but was extremely lacklustre in the quality of the data or additionally difficult to scrape/obtain. Hence they were skipped for the purposes of using as this project's data source.

- https://www.hwcompare.com/
- https://www.userbenchmark.com/Software
- https://www.tomshardware.com/reviews/gpu-hierarchy,4388.html
- https://www.tomshardware.com/reviews/cpu-hierarchy,4312.html

To view some of these sample data, it is located under `./data/research_data`

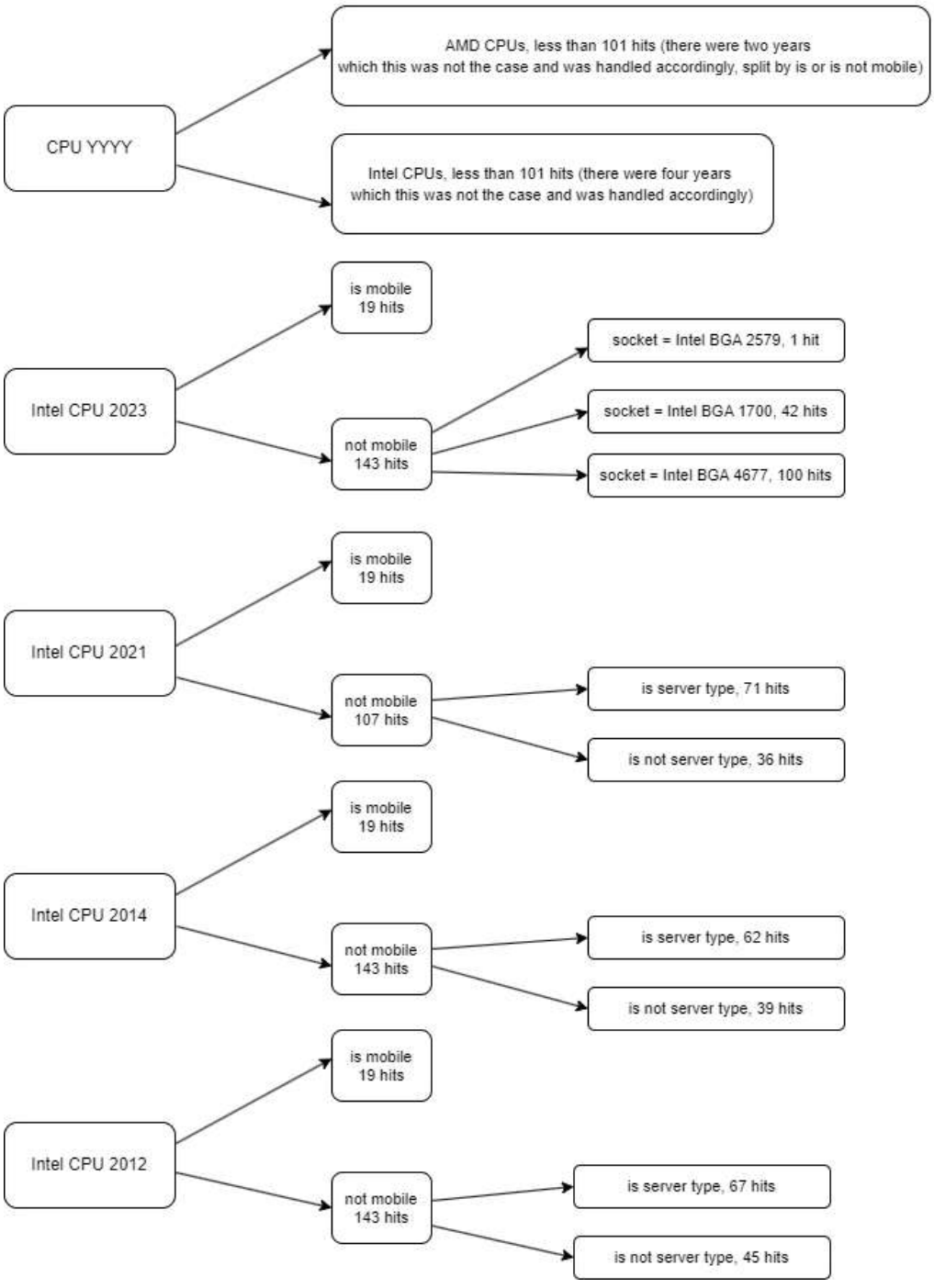
I have included diagrams illustrating the web scraping methodology used to collect a comprehensive dataset from TechPowerUp below:

Data Scraping Methodology

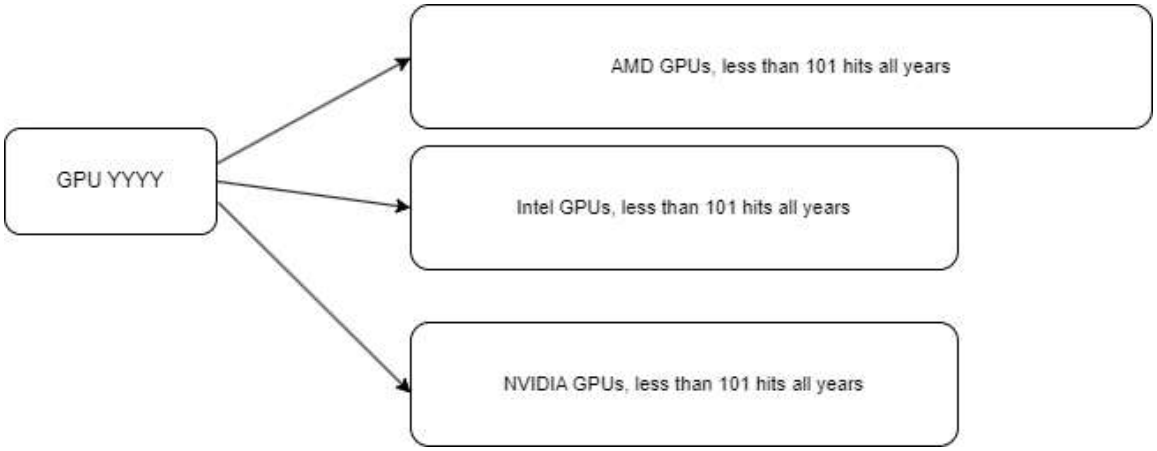
Data Scraping Workflow - CPU Filtering

Techpowerup website has a limit of 100 results per query. Hence I needed to filter and grab various endpoint queries to obtain **ALL** cpu data.

Example url might looks like: https://www.techpowerup.com/cpu-specs/?mfgr=AMD&released=2023&mobile=No&sort=name

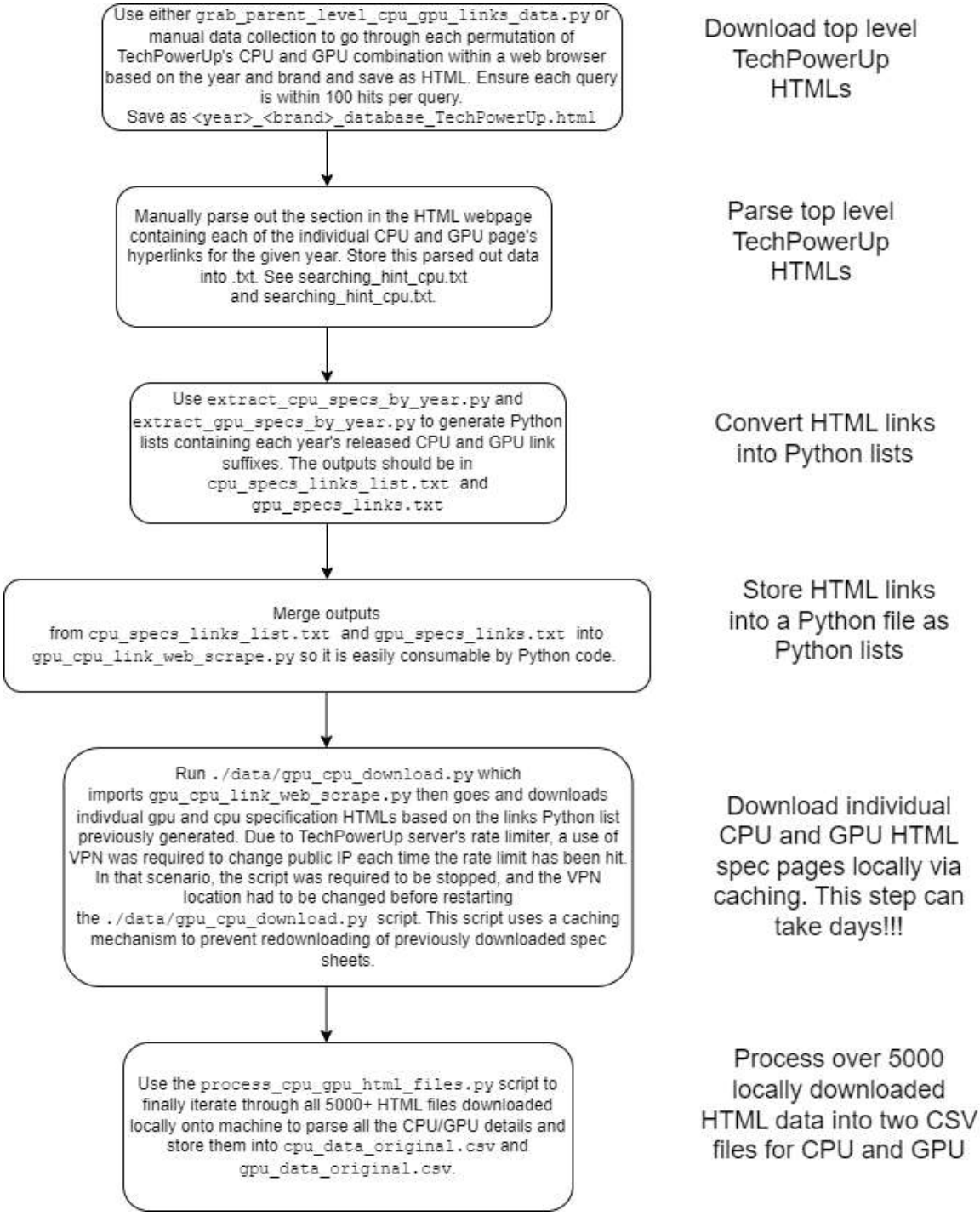


Data Scraping Workflow - GPU Filtering



Please note that for GPU data collection, filtering was applied by brand (AMD, Intel, Nvidia) only.

Data Scraping Full Workflow



<!--

<b -->r>

) only.

v>

Please note that the website rate limits how much data you can scrape in a day. Hence the use of VPN to change the public IP address was required (to download over 5000 HTML

files in a few days).

After completing the pipeline above, we are provided with an invaluable dataset containing 21 years (2004 to 2024) of CPU and GPU data. They are generally standard CPU and GPU specification related data as well as its release date and other data consumers are generally interested about.

In [1]:
```
import pandas as pd

# Set display options to show all columns
pd.set_option('display.max_columns', None)  # Show all columns
pd.set_option('display.expand_frame_repr', False)  # Disable column wrapping

# Relative paths to the CSV files
cpu_data_path = '../src/cpu_data_original.csv'
gpu_data_path = '../src/gpu_data_original.csv'

# Load the data
cpu_data = pd.read_csv(cpu_data_path)
gpu_data = pd.read_csv(gpu_data_path)

# Display the headers and first few rows for CPU data
print("CPU Data - Header and First 5 Rows:")
display(cpu_data.head())

# Display the headers and first few rows for GPU data
print("\nGPU Data - Header and First 5 Rows:")
display(gpu_data.head())
```

```
CPU Data - Header and First 5 Rows:
```

	Name	Physical - Socket:	Physical - Foundry:	Physical - Process Size:	Physical - Transistors:	Physical - Die Size:	Physical - Package:	Physic tCaseM
0	AMD 4800S Specs | TechPowerUp CPU Database	AMD BGA 2963	TSMC	7 nm	15,300 million	360 mm²	FC-BGA	1(
1	AMD A10-4600M Specs | TechPowerUp CPU Database	AMD Socket FS1r2	GlobalFoundries	32 nm	1,303 million	246 mm²	µPGA	N
2	AMD A10-4655M Specs | TechPowerUp CPU Database	AMD Socket FP2	GlobalFoundries	32 nm	1,303 million	246 mm²	BGA2	1(
3	AMD A10-5700 Specs | TechPowerUp CPU Database	AMD Socket FM2	GlobalFoundries	32 nm	1,303 million	246 mm²	µPGA	7
4	AMD A10-5745M Specs | TechPowerUp CPU Database	AMD Socket FP2	NaN	32 nm	1,178 million	246 mm²	BGA2	7

GPU Data - Header and First 5 Rows:

	GPU Name	GPU Variant	Architecture	Foundry	Process Size	Transistors	Density	Die Size	Chip Package	Re
0	GA102	GA102-890-A1	Ampere	Samsung	8 nm	28,300 million	45.1M / mm²	628 mm²	BGA-3328	
1	GA100	NaN	Ampere	TSMC	7 nm	54,200 million	65.6M / mm²	826 mm²	BGA-2743	
2	GA100	NaN	Ampere	TSMC	7 nm	54,200 million	65.6M / mm²	826 mm²	BGA-2743	
3	GA100	NaN	Ampere	TSMC	7 nm	54,200 million	65.6M / mm²	826 mm²	BGA-2743	
4	GA100	NaN	Ampere	TSMC	7 nm	54,200 million	65.6M / mm²	826 mm²	BGA-2743	

Data Description

CPU Data

Below are some key facts about the `cpu_data_original.csv` dataset:

Description	Value
File Size	1,744 KB
Total Number of Rows	3,460
Total Number of Columns	174
CPUs Released in 2004	146
CPUs Released in 2005	167
CPUs Released in 2006	172

Description	Value
CPUs Released in 2007	91
CPUs Released in 2008	161
CPUs Released in 2009	129
CPUs Released in 2010	159
CPUs Released in 2011	210
CPUs Released in 2012	257
CPUs Released in 2013	240
CPUs Released in 2014	158
CPUs Released in 2015	122
CPUs Released in 2016	65
CPUs Released in 2017	128
CPUs Released in 2018	142
CPUs Released in 2019	140
CPUs Released in 2020	113
CPUs Released in 2021	173
CPUs Released in 2022	142
CPUs Released in 2023	297
CPUs Released in 2024	123
CPUs Without Release Date	124

Note: Please refer to the table in the previous section for the different CPU columns. Generally, the more important and less sparse datasets are found on the left-hand side of the table.

GPU Data

Below are some key facts about the `gpu_data_original.csv` dataset:

Description	Value
File Size	1,477 KB
Total Number of Rows	2,650
Total Number of Columns	154
GPUs Released in 2004	59
GPUs Released in 2005	36

Description	Value
GPUs Released in 2006	89
GPUs Released in 2007	73
GPUs Released in 2008	109
GPUs Released in 2009	62
GPUs Released in 2010	94
GPUs Released in 2011	167
GPUs Released in 2012	172
GPUs Released in 2013	206
GPUs Released in 2014	146
GPUs Released in 2015	162
GPUs Released in 2016	108
GPUs Released in 2017	113
GPUs Released in 2018	94
GPUs Released in 2019	108
GPUs Released in 2020	89
GPUs Released in 2021	100
GPUs Released in 2022	88
GPUs Released in 2023	93
GPUs Released in 2024	14
GPUs Without Release Date	468

Note: Please refer to the table in the previous section for the different GPU columns. Generally, the more important and less sparse datasets are found on the left-hand side of the table. ss sparse datasets are found on the left-hand side of the table.

Data Cleaning

An initial observation of the `cpu_data_original.csv` and `gpu_data_original.csv` datasets reveals that data cleaning is necessary to prepare the data for analysis and modeling. Certain inconsistencies and redundant information need to be addressed to ensure the datasets are in a clean, usable format.

Key Findings:

CPU Data

The following table highlights the key cleaning tasks required in the `cpu_data_original.csv` file, along with the reasons why these cleanups are needed (note extra cleaning were done but the ones below were the key ones):

What	Reason for Cleaning Task
"Name" column has "Specs | TechPowerUp CPU Database" suffix	To make processing in Python easier, consistent and for clarity, remove it
Various data columns with empty entries	Replace the empty entries with "Not Provided" to make data processing easier in Python.
"Physical - Die Size", "Physical - Package", "Physical - tCaseMax" has strange characters	Remove strange characters like Â from the dataset
Release Date column has inconsistent formats	Standarize the date formatting so it will be easier to process with Python
Various numerical columns like "Physical - Process Size", "Physical - Transistors", "Performance - Frequency" etc has units	Standarize the numerical values by adding the units in the Column header and standarizing the values with a given unit.
Splitting Cache related descriptors into multiple granular columns for better analysis	Very challenging to work with data that can look like "512 KB (per core)" or "1 MB (shared)"
Adding a new column that is inflation adjusted launch MSRP. The inflation was adjusted using the data from https://www.usinflationcalculator.com/inflation/current-inflation-rates/.	To get insight into the general rising cost trend of products, this metric is necessary.

The same process of cleaning will be applied to other columns where similar issues exist. The final cleaned data can be found in `cpu_data_cleaned.xls` .

GPU Data

The following table highlights the key cleaning tasks required in the `gpu_data_original.csv` file, along with the reasons why these cleanups are needed (note extra cleaning were done but the ones below were the key ones):

What	Reason for Cleaning Task

"Name" column has "Specs | TechPowerUp CPU Database" suffix and in middle of the spreadsheet	To make processing in Python easier, consistent and for clarity, remove the suffix and move the column to the front
Various data columns with empty entries	Replace the empty entries with "Not Provided" to make data processing easier in Python.
"Die Size", "Package Size", "Physical - tCaseMax" etc has strange characters	Remove strange characters like Â from the dataset
Release Date column has inconsistent formats	Standarize the date formatting so it will be easier to process with Python
Various numerical columns like "Base Clock", "Transistors", "Memory Size" etc has units	Standarize the numerical values by adding the units in the Column header and standarizing the values with a given unit.
Splitting Cache, Memory Clock and other columns which has various data bundled into one column. Requires unpacking said columns into various related descriptor columns for better analysis	Very challenging to work with data that can look like "512 KB (per core)" or "1593 MHz3.2 Gbps effective"
Adding a new column that is inflation adjusted launch MSRP. The inflation was adjusted using the data from https://www.usinflationcalculator.com/inflation/current-inflation-rates/.	To get insight into the general rising cost trend of products, this metric is necessary.

The same process of cleaning will be applied to other columns where similar issues exist. The final cleaned data can be found in `gpu_data_cleaned.xls`.

In summary, the data cleaning process was essential to transform both the `cpu_data_original.csv` and `gpu_data_original.csv` datasets into a consistent, analyzable format. By addressing empty values, inconsistent units, and strange characters, the data was standardized for easier processing. Key challenges included handling columns with multiple bundled data types, such as cache and clock speeds, which required splitting into granular columns for clarity. Additionally, inflation-adjusted MSRP values were calculated to provide historical context for price trends. Given that the data originates from TechPowerUp.com, one of the most reliable PC hardware information websites, additional visualizations for identifying anomalies or outliers were not necessary.

Data Visualization Post Cleaning

In [2]:
```
import pandas as pd

# Set display options to show all columns
pd.set_option('display.max_columns', None)  # Show all columns
pd.set_option('display.expand_frame_repr', False)  # Disable column wrapping

# Relative paths to the Excel files
cpu_data_path = '../src/cpu_data_cleaned.xlsx'
gpu_data_path = '../src/gpu_data_cleaned.xlsx'

# Load the data using read_excel since the files are .xls
cpu_data = pd.read_excel(cpu_data_path)
gpu_data = pd.read_excel(gpu_data_path)

# Display the headers and first few rows for CPU data
print("CPU Data - Header and First 5 Rows:")
display(cpu_data.head())

# Display the headers and first few rows for GPU data
print("\nGPU Data - Header and First 5 Rows:")
display(gpu_data.head())
```

```
CPU Data - Header and First 5 Rows:
```

	Name	Physical - Socket:	Physical - Foundry	Physical - Process Size (in nm)	Physical - Transistors (in millions)	Physical - Die Size(in mm²)	Physical - Package	Physical - tCaseMax (in °C)	Pr -
0	AMD 4800S	AMD BGA 2963	TSMC	7	15300	360	FC-BGA	100	
1	AMD A10-4600M	AMD Socket FS1r2	GlobalFoundries	32	1303	246	µPGA	Not Provided	
2	AMD A10-4655M	AMD Socket FP2	GlobalFoundries	32	1303	246	BGA2	100	
3	AMD A10-5700	AMD Socket FM2	GlobalFoundries	32	1303	246	µPGA	71	
4	AMD A10-5745M	AMD Socket FP2	Not Provided	32	1178	246	BGA2	71	

```
GPU Data - Header and First 5 Rows:
```

	Name	GPU Name	GPU Variant	Architecture	Foundry	Process Size (in nm)	Transistors (in millions)	Density (in M / mm²)	Die Size (in mm²)	F
0	NVIDIA A10 PCIe	GA102	GA102-890-A1	Ampere	Samsung	8	28300	45.1	628	
1	NVIDIA A100 PCIe 40 GB	GA100	Not Provided	Ampere	TSMC	7	54200	65.6	826	
2	NVIDIA A100 PCIe 80 GB	GA100	Not Provided	Ampere	TSMC	7	54200	65.6	826	
3	NVIDIA A100 SXM4 40 GB	GA100	Not Provided	Ampere	TSMC	7	54200	65.6	826	
4	NVIDIA A100 SXM4 80 GB	GA100	Not Provided	Ampere	TSMC	7	54200	65.6	826	

Exploratory Data Analysis

In this section, we conduct an in-depth exploration of the data to uncover key characteristics and trends. By analyzing these features, we can identify the most relevant attributes for building our supervised machine learning models to predict CPU and GPU prices.

Recall that the primary goals of this project are:

1. What are the trends in CPU and GPU parts over the past 20 years?
2. Is the price-to-performance ratio of these parts keeping up? Are consumers getting a fair deal compared to 10 to 20 years ago?
3. Can we predict the performance of next-gen, unreleased CPU and GPU parts using supervised machine learning models? models?

In this section, we will address the first two questions through visualizations and data analysis, offering insights into historical trends and the evolving price-performance

relationship. Additionally, we will perform feature engineering to select the most significant features for training machine learning models aimed at accurately predicting CPU and GPU prices. By understanding the underlying patterns and refining our dataset, we will be better equipped to build reliable predictive models in the next phase of this project.

A simple internet search on what makes a CPU or GPU "good" helps us pinpoint the best features to focus on.

For CPUs, the following features make CPU good (there are countless others but we will stick with this subset):

1. **Frequency (GHz)** - Higher the better
2. **Number of Cores/Threads** - More the better (in general)
3. **Cache Size (L1, L2, L3)** - Larger the better
4. **Thermal Design Power (TDP)** - More power consumption indicates a more powerful CPU
5. **Integrated Graphics** - Having more cost more to produce
6. **Transistor Count** - more transistors means more expensive
7. **Process Size (nm)** - smaller process nodes indicate more cost to manufacture

For GPUs, the following features make GPU "good" (there are countless others but we will stick with this subset):

1. **Frequency (GHz)** - Higher the better
2. **Memory Size** - More the better
3. **Memory Type** - Newer models the better
4. **Effective Speed** - Faster the better
5. **Memory Bus** - Larger the better
6. **Bandwidth (in GB/s)** - Higher the better
7. **Shading Units (CUDA cores)** - More the better
8. **TMUs (Texture Mapping Units)** - More the better
9. **ROPs (Render Output Units)** - More the better
10. **SM Count (Streaming Multiprocessors)** - More the better
11. **Tensor Cores** - More the better
12. **RT Cores** - More the better
13. **Cache Size** - Larger is better
14. **Pixel Rate (in GPixel/s)** - Higher the better
15. **Texture Rate (in GTexel/s)** - Higher the better
16. **FP Computation** - Higher the better
17. **TDP (in Watts)** - Higher power often means better performance
18. **Transistor Count** - More transistors means more expensive
19. **Process Size (nm)** - Smaller process nodes indicate more cost to manufacture

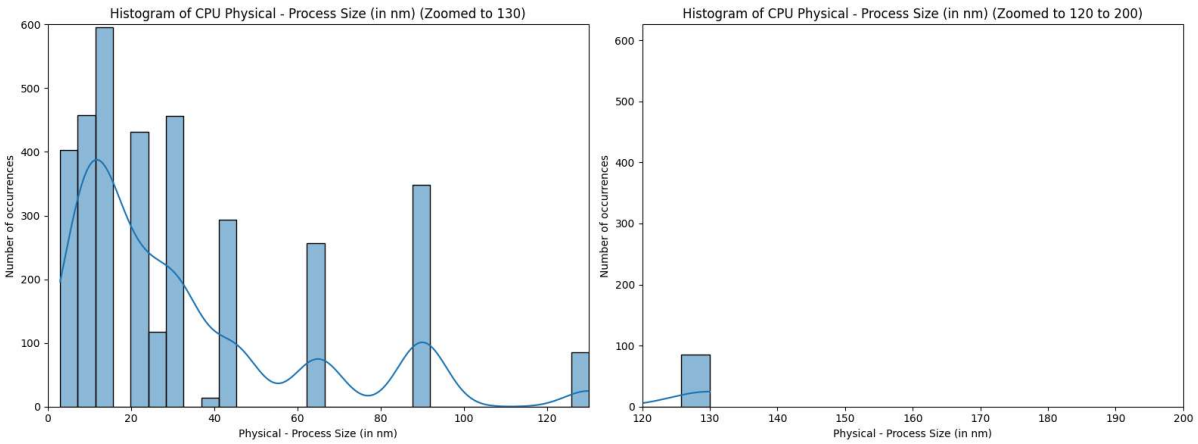
Sources:

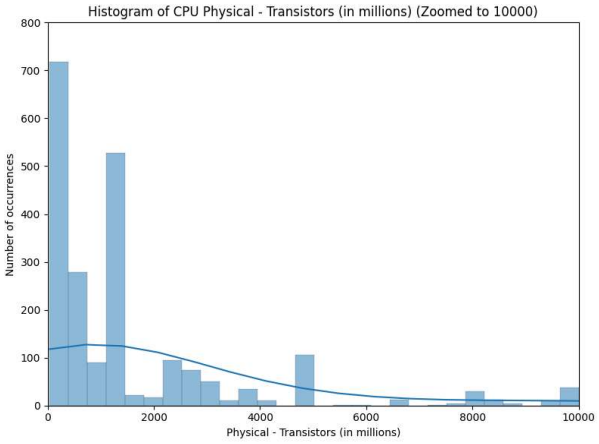
- https://www.hp.com/us-en/shop/tech-takes/what-is-processor-speed
- https://www.intel.com/content/www/us/en/gaming/resources/gaming-cpu.html
- https://www.tomshardware.com/reviews/cpu-buying-guide,5643.html
- https://www.newegg.com/insider/how-to-choose-a-cpu/
- https://www.hp.com/us-en/shop/tech-takes/gpu-buying-guide
- https://www.tomshardware.com/reviews/gpu-buying-guide,5844.html
- https://www.gamecrate.com/hardware/how-to-choose-best-graphics-card-pc-gaming

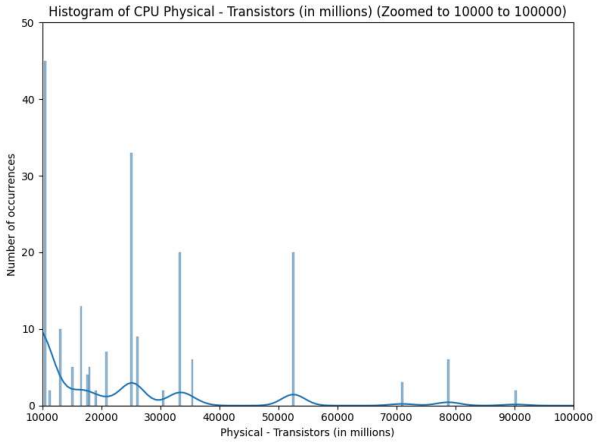
In [3]:
```
import math
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Load the CPU data
cpu_data = pd.read_excel('../src/cpu_data_cleaned.xlsx')


cpu_data.columns = cpu_data.columns.str.strip()
# List of CPU columns for histograms (numerical columns)
cpu_numerical_columns = [
    'Physical - Process Size (in nm)',
    'Physical - Transistors (in millions)',
    'Physical - Die Size(in  mm²)',
    'Performance - Frequency (in GHz)',
    'Performance - Turbo Clock (up to in GHz)',
    'Performance - TDP (in Watts)',
    'Core Config - # of Cores',
    'Core Config - # of Threads',
    'Cache L1 Size (in KB)',
    'Cache L2 Size (in MB)',
    'Cache L3 Size (in MB)'
]

# List of CPU categorical columns
cpu_categorical_columns = [
    'Architecture - Memory Bus',
    'Core Config - Integrated Graphics',
    "Processor - Market",
    "Processor - Production Status",
]

# Function to convert non-numeric values to NaN
def convert_to_numeric(data, columns):
    for column in columns:
        if column in data.columns:
            data[column] = pd.to_numeric(data[column], errors='coerce')  # Convert

def convert_integrated_graphics_to_boolean(data, column):
    if column in data.columns:
        # Convert to boolean: True if there's a value (i.e. the CPU has integrated
        data[column] = data[column].notna()
```

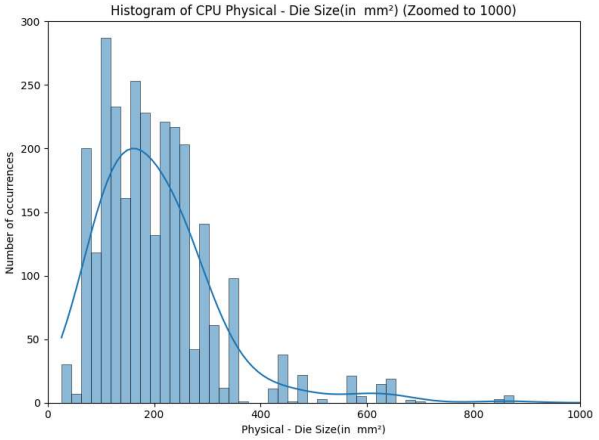
```
# Convert the numerical columns to numeric (handle non-numeric values)
convert_to_numeric(cpu_data, cpu_numerical_columns)
# Convert the "Core Config - Integrated Graphics" column to boolean
convert_integrated_graphics_to_boolean(cpu_data, 'Core Config - Integrated Graphic


# Create histograms with custom limits for each feature
def plot_histograms_with_custom_limits(data, columns_with_limits):
    # Iterate over each column and its custom limits
    for column, limits in columns_with_limits.items():
        xlim_max1, xlim_max2, ylim_max1, ylim_max2, zoom_range = limits

        # First plot with xlim_max1 and ylim_max1 for both subplots
        plt.figure(figsize=(16, 6))

        # Plot 1: with xlim_max1 and ylim_max1
        plt.subplot(1, 2, 1)
        sns.histplot(data[column].dropna(), kde=True)
        plt.title(f'Histogram of CPU {column} (Zoomed to {xlim_max1})')
        plt.xlabel(column)
        plt.ylabel('Number of occurrences')
        if xlim_max1 is not None:
            plt.xlim(0, xlim_max1)
        if ylim_max1 is not None:
            plt.ylim(0, ylim_max1)

        # Plot 2: with xlim_max2 and ylim_max2, narrower zoom range, and no binwi
        plt.subplot(1, 2, 2)
        sns.histplot(data[column].dropna(), kde=True)
        plt.title(f'Histogram of CPU {column} (Zoomed to {zoom_range[0]} to {zoom_
        plt.xlabel(column)
        plt.ylabel('Number of occurrences')
        plt.xlim(zoom_range[0], zoom_range[1])
        if ylim_max2 is not None:
            plt.ylim(0, ylim_max2)

        plt.tight_layout()
        plt.show()



# Create bar plots for categorical CPU columns
def plot_categorical_barplots(data, categorical_columns):
    # Calculate the number of rows needed, with 2 plots per row
    n_cols = 2
    n_rows = math.ceil(len(categorical_columns) / n_cols)

    # Create a figure for subplots
    fig, axes = plt.subplots(n_rows, n_cols, figsize=(16, n_rows * 6))
    axes = axes.flatten()  # Flatten the axes for easy iteration

    # Plot each categorical column in its respective subplot
    for i, column in enumerate(categorical_columns):
        if column in data.columns:
            sns.countplot(y=data[column], order=data[column].value_counts().index
            axes[i].set_title(f'Bar Plot of CPU {column}')
```

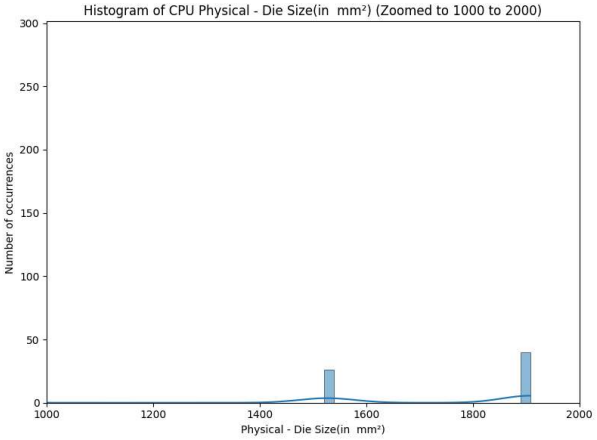
```
            axes[i].set_xlabel('Number of occurrences')
            axes[i].set_ylabel(column)

    # Hide any remaining empty subplots
    for j in range(i + 1, len(axes)):
        fig.delaxes(axes[j])

    # Adjust layout to prevent overlap and add space between columns
    plt.tight_layout()
    plt.subplots_adjust(wspace=0.4, hspace=0.3)  # Increase space between plots

    plt.show()

# Call the functions to plot histograms and bar plots

columns_with_limits = {
    'Physical - Process Size (in nm)': (130, None, 600, None, (120, 200)),
    'Physical - Transistors (in millions)': (10000, 150000, 800, 50, (10000, 10000
    'Physical - Die Size(in  mm²)': (1000, None, 300, None, (1000, 2000)),
    'Performance - Frequency (in GHz)': (5, None, 500, 60, (4, 10)),
    'Performance - Turbo Clock (up to in GHz)': (5, None, 500, 200, (4, 10)),
    'Performance - TDP (in Watts)': (500, None, 1000, None, (10, 500)),
    'Core Config - # of Cores': (20, None, 1000, 75, (20, 140)),
    'Core Config - # of Threads': (150, None, 1500, 300, (150, 300)),
    'Cache L1 Size (in KB)': (200, None, 2000, 90, (200, 800)),
    'Cache L2 Size (in MB)': (15, None, 1250, 5, (15, 511)),
    'Cache L3 Size (in MB)': (200, None, 1000, 50, (200, 1200)),
}

# Call the function to plot histograms with custom limits
plot_histograms_with_custom_limits(cpu_data, columns_with_limits)

plot_categorical_barplots(cpu_data, cpu_categorical_columns)
```

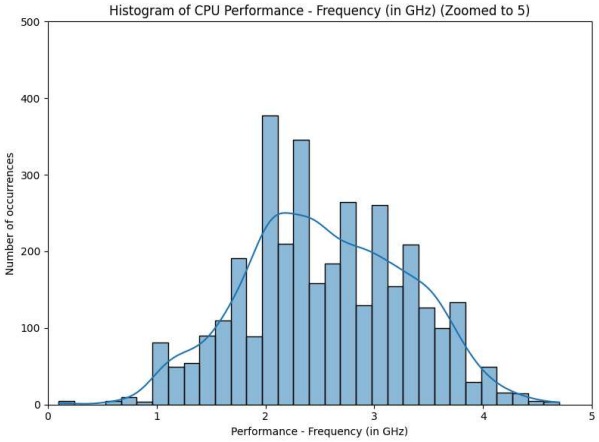


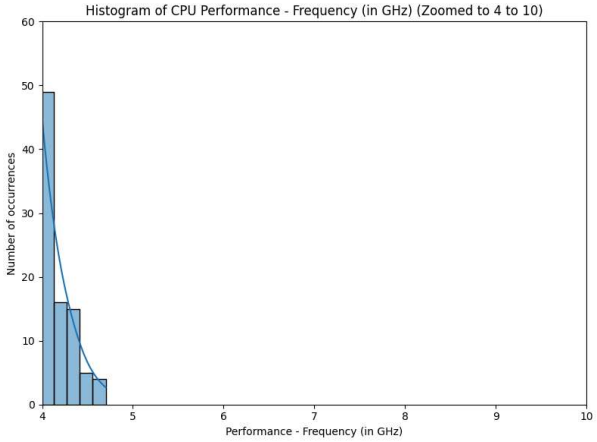

Histogram of CPU Physical - Transistors (in millions) (Zoomed to 10000)
Number of occurrences
Physical - Transistors (in millions)

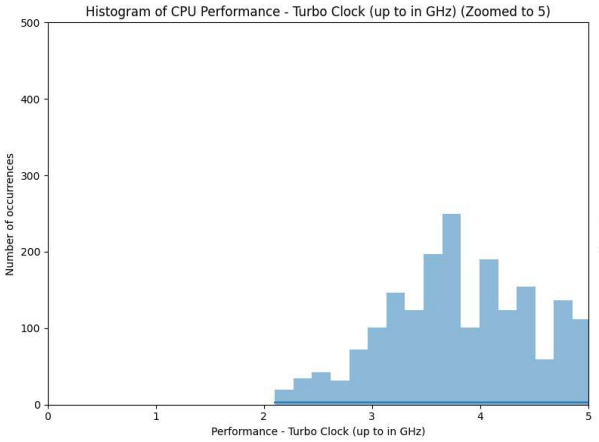

Histogram of CPU Physical - Transistors (in millions) (Zoomed to 10000 to 100000)
Number of occurrences
Physical - Transistors (in millions)

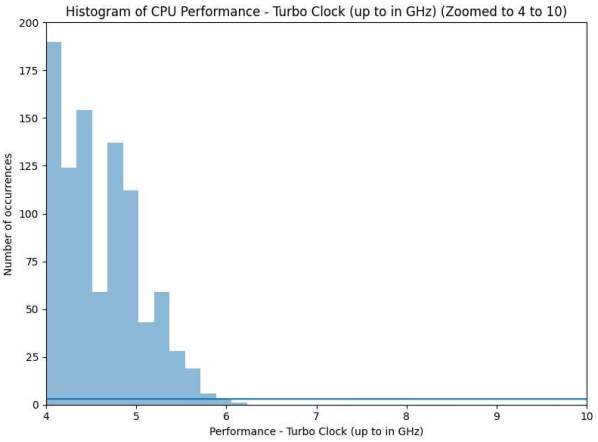

Histogram of CPU Physical - Die Size(in mm²) (Zoomed to 1000)
Number of occurrences
Physical - Die Size(in mm²)

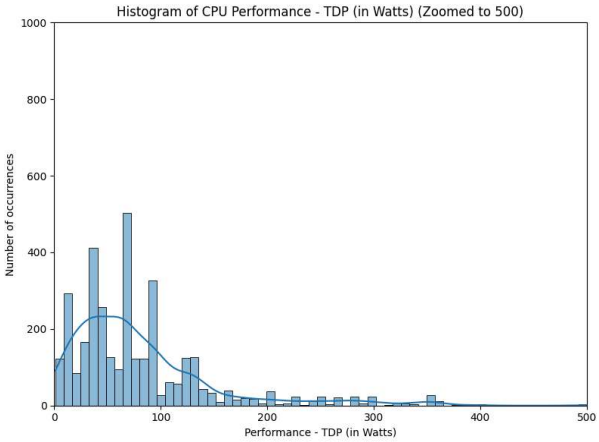

Histogram of CPU Physical - Die Size(in mm²) (Zoomed to 1000 to 2000)
Number of occurrences
Physical - Die Size(in mm²)

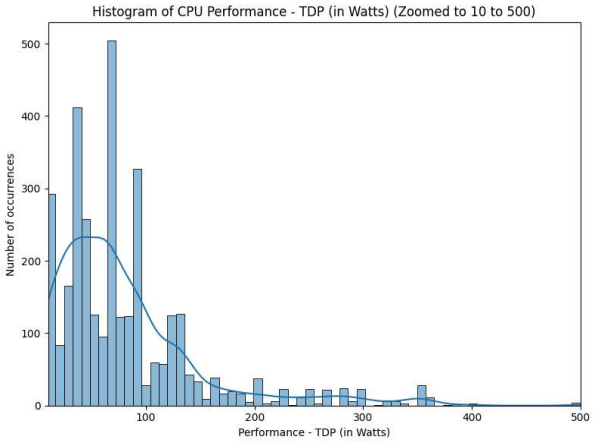

Histogram of CPU Performance - Frequency (in GHz) (Zoomed to 5)
Number of occurrences
Performance - Frequency (in GHz)

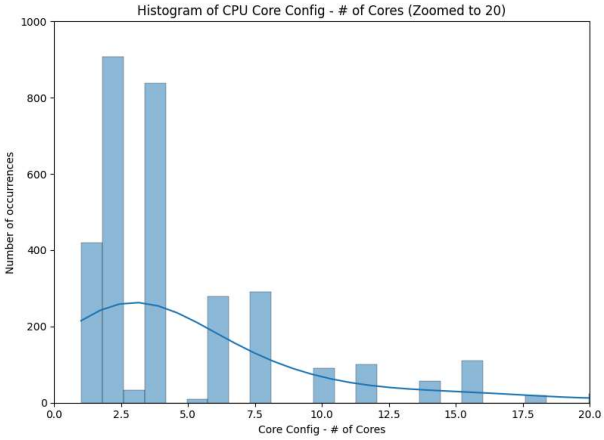

Histogram of CPU Performance - Frequency (in GHz) (Zoomed to 4 to 10)
Number of occurrences
Performance - Frequency (in GHz)

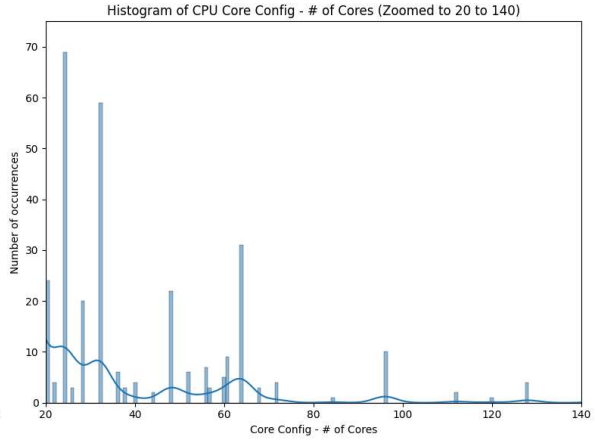

Histogram of CPU Performance - Turbo Clock (up to in GHz) (Zoomed to 5)
Number of occurrences
Performance - Turbo Clock (up to in GHz)

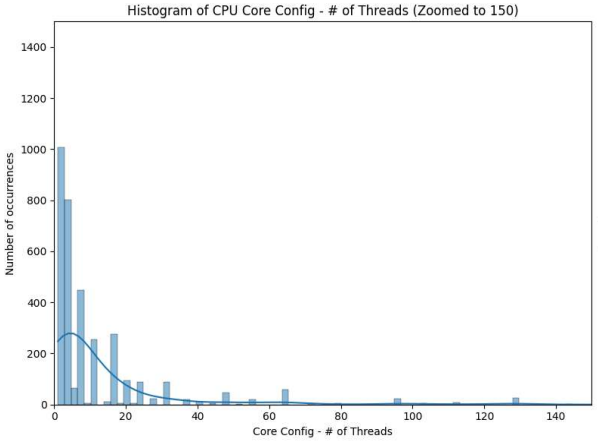

Histogram of CPU Performance - Turbo Clock (up to in GHz) (Zoomed to 4 to 10)
Number of occurrences
Performance - Turbo Clock (up to in GHz)

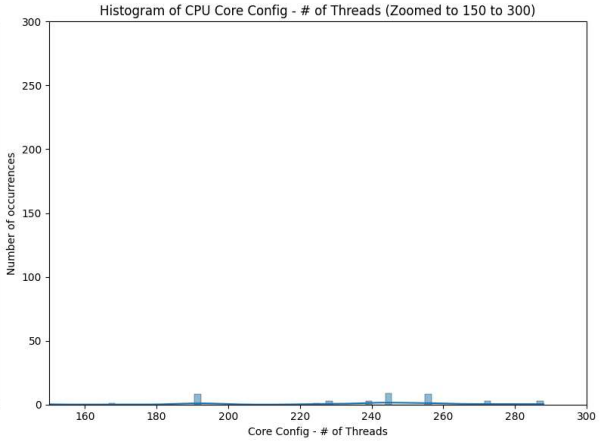

Histogram of CPU Performance - TDP (in Watts) (Zoomed to 500)
Number of occurrences
Performance - TDP (in Watts)

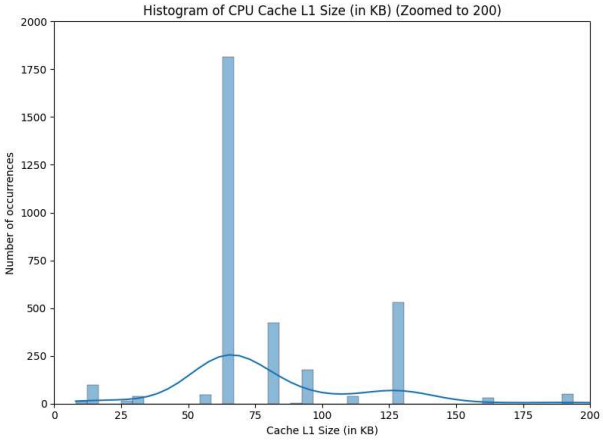

Histogram of CPU Performance - TDP (in Watts) (Zoomed to 10 to 500)
Number of occurrences
Performance - TDP (in Watts)

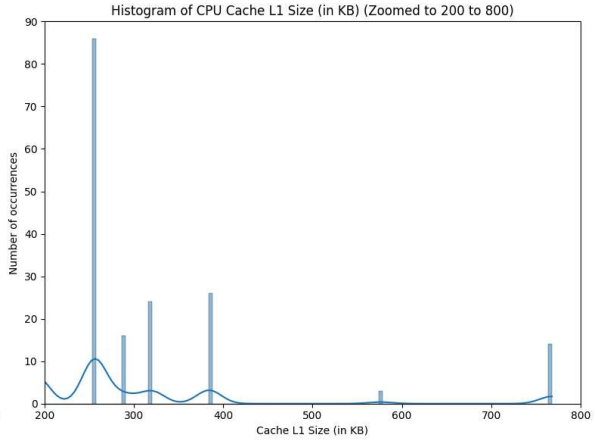

Histogram of CPU Core Config - # of Cores (Zoomed to 20)
Number of occurrences
Core Config - # of Cores

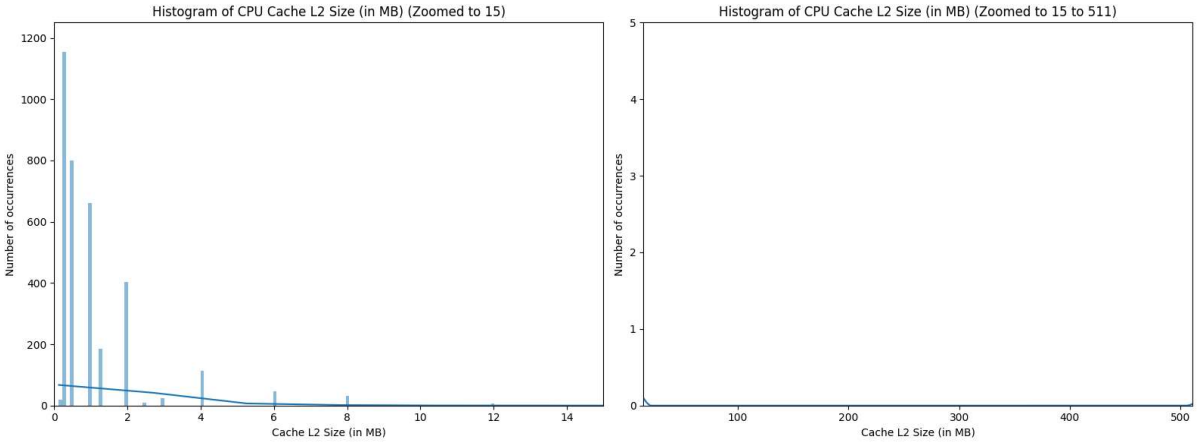

Histogram of CPU Core Config - # of Cores (Zoomed to 20 to 140)
Number of occurrences
Core Config - # of Cores

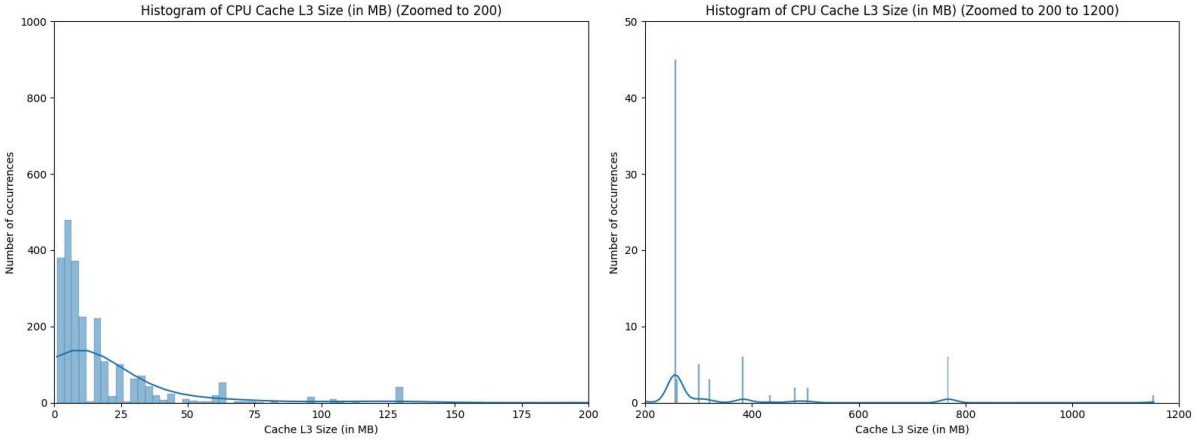

Histogram of CPU Core Config - # of Threads (Zoomed to 150)
Number of occurrences
Core Config - # of Threads

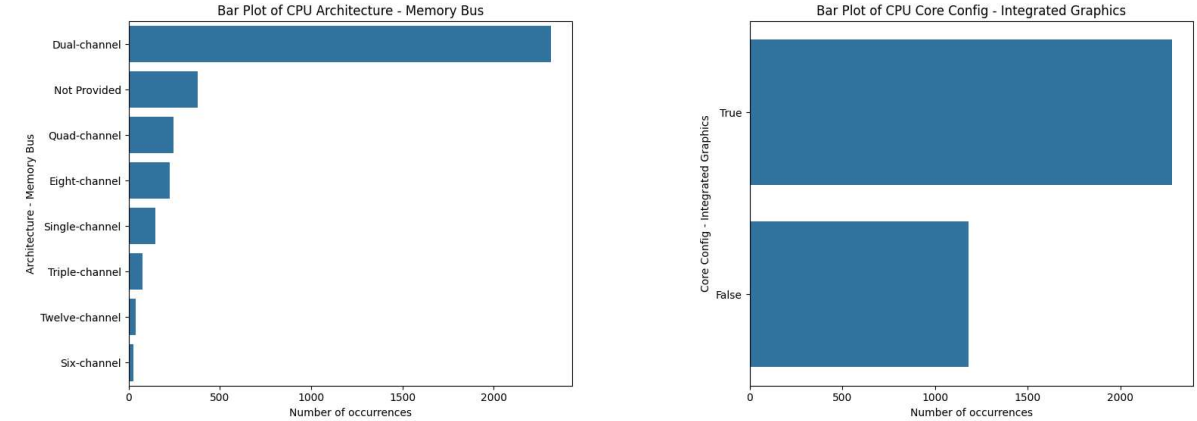

Histogram of CPU Core Config - # of Threads (Zoomed to 150 to 300)
Number of occurrences
Core Config - # of Threads

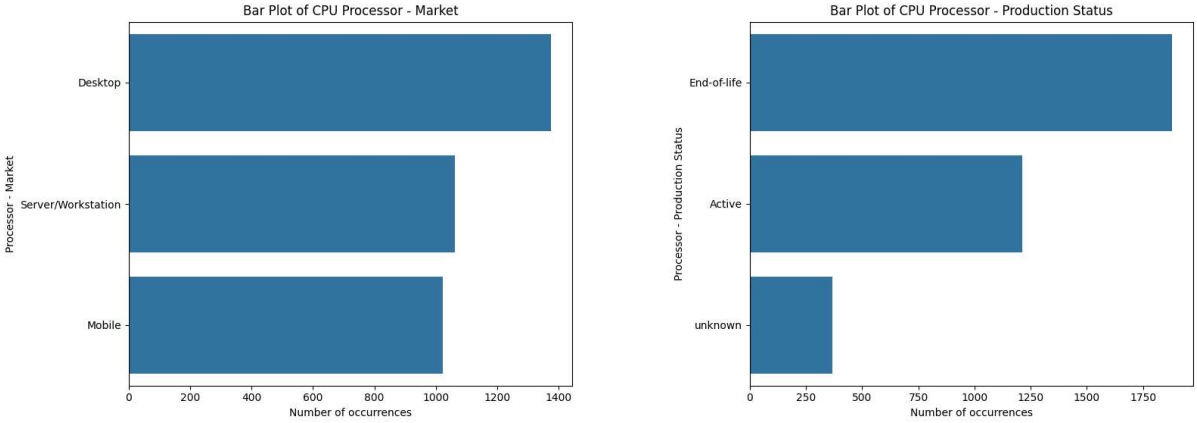

Histogram of CPU Cache L1 Size (in KB) (Zoomed to 200)
Number of occurrences
Cache L1 Size (in KB)


Histogram of CPU Cache L1 Size (in KB) (Zoomed to 200 to 800)
Number of occurrences
Cache L1 Size (in KB)







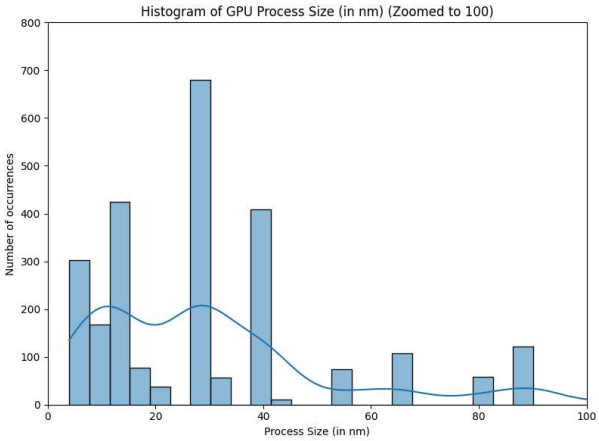


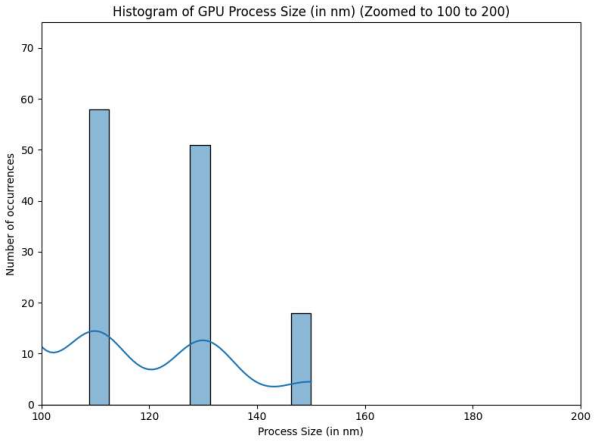
In [4]:
```
import pandas as pd
import matplotlib.pyplot as plt
```

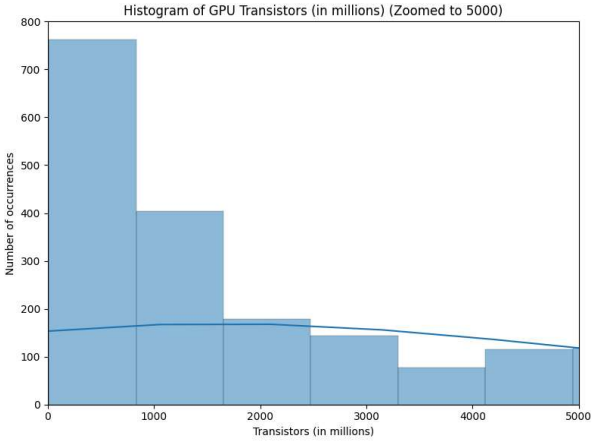
```
import seaborn as sns
import math

# Load the GPU data
gpu_data = pd.read_excel('../src/gpu_data_cleaned.xlsx')

# Strip whitespace from column names
gpu_data.columns = gpu_data.columns.str.strip()

# List of GPU columns for histograms (numerical columns)
gpu_numerical_columns = [
    "Process Size (in nm)",
    "Transistors (in millions)",
    "Density (in M / mm²)",
    "Die Size (in mm²)",
    "Base Clock (in MHz)",
    "Boost Clock (in MHz)",
    "Memory Clock (in MHz)",
    "Effective Speed (in Gbps)",
    "Memory Size (in GB)",
    "Memory Bus (in bit)",
    "Bandwidth (in GB/s)",
    "Shading Units",
    "TMUs",
    "ROPs",
    "SM Count",
    "Tensor Cores",
    "RT Cores",
    "L1 Cache (in KB)",
    "L2 Cache (in MB)",
    "Pixel Rate (in GPixel/s)",
    "Texture Rate (in Gtexel/s)",
    "FP16 (half, in TFLOPS)",
    "FP32 (float, in TFLOPS)",
    "FP64 (double, in TFLOPS)",
    "TDP (in Watts)"
]

# List of GPU categorical columns
gpu_categorical_columns = [
    "Production"
]

# Function to convert non-numeric values to NaN
def convert_to_numeric(data, columns):
    for column in columns:
        if column in data.columns:
            data[column] = pd.to_numeric(data[column], errors='coerce')

# Convert the numerical columns to numeric (handle non-numeric values)
convert_to_numeric(gpu_data, gpu_numerical_columns)

# Create histograms with custom limits for each feature
def plot_histograms_with_custom_limits(data, columns_with_limits):
    # Iterate over each column and its custom limits
    for column, limits in columns_with_limits.items():
```

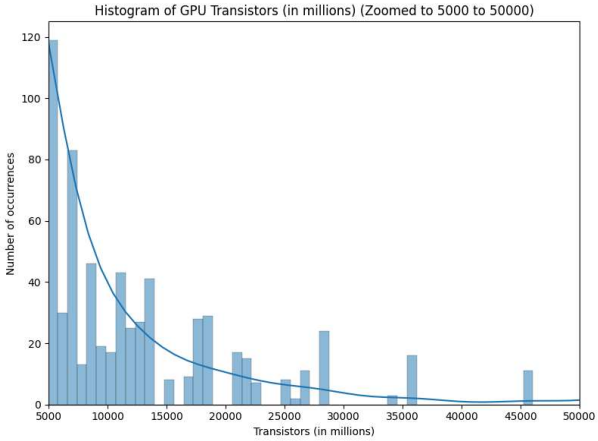
```
        xlim_max1, xlim_max2, ylim_max1, ylim_max2, zoom_range = limits

        # First plot with xlim_max1 and ylim_max1 for both subplots
        plt.figure(figsize=(16, 6))

        # Plot 1: with xlim_max1 and ylim_max1
        plt.subplot(1, 2, 1)
        sns.histplot(data[column].dropna(), kde=True)
        plt.title(f'Histogram of GPU {column} (Zoomed to {xlim_max1})')
        plt.xlabel(column)
        plt.ylabel('Number of occurrences')
        if xlim_max1 is not None:
            plt.xlim(0, xlim_max1)
        if ylim_max1 is not None:
            plt.ylim(0, ylim_max1)

        # Plot 2: with xlim_max2 and ylim_max2, narrower zoom range, and no binwid
        plt.subplot(1, 2, 2)
        sns.histplot(data[column].dropna(), kde=True)
        plt.title(f'Histogram of GPU {column} (Zoomed to {zoom_range[0]} to {zoom_
        plt.xlabel(column)
        plt.ylabel('Number of occurrences')
        plt.xlim(zoom_range[0], zoom_range[1])
        if ylim_max2 is not None:
            plt.ylim(0, ylim_max2)

        plt.tight_layout()
        plt.show()

# Create bar plots for categorical GPU columns
def plot_categorical_barplots(data, categorical_columns):
    # Calculate the number of rows needed, with 2 plots per row
    n_cols = 2
    n_rows = math.ceil(len(categorical_columns) / n_cols)

    # Create a figure for subplots
    fig, axes = plt.subplots(n_rows, n_cols, figsize=(16, n_rows * 6))
    axes = axes.flatten()  # Flatten the axes for easy iteration

    # Plot each categorical column in its respective subplot
    for i, column in enumerate(categorical_columns):
        if column in data.columns:
            sns.countplot(y=data[column], order=data[column].value_counts().index
            axes[i].set_title(f'Bar Plot of GPU {column}')
            axes[i].set_xlabel('Number of occurrences')
            axes[i].set_ylabel(column)

    # Hide any remaining empty subplots
    for j in range(i + 1, len(axes)):
        fig.delaxes(axes[j])

    # Adjust layout to prevent overlap and add space between columns
    plt.tight_layout()
    plt.subplots_adjust(wspace=0.4, hspace=0.3)  # Increase space between plots

    plt.show()
```

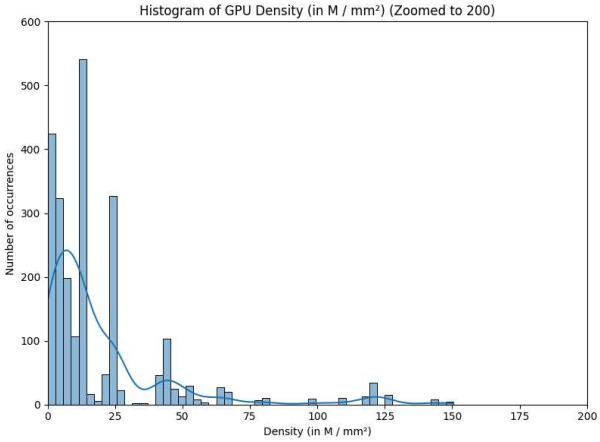
```
# Call the functions to plot histograms and bar plots
columns_with_limits = {
    'Process Size (in nm)': (100, None, 800, 75, (100, 200)), #done
    'Transistors (in millions)': (5000, 30000, 800, 125, (5000, 50000)), #done
    'Density (in M / mm²)': (200, None, 600, 50, (200, 500)), #done
    'Die Size (in mm²)': (850, None, 450, 30, (800, 2000)), #done
    'Base Clock (in MHz)': (2500, None, 200, 60, (2400, 5000)), #done
    'Boost Clock (in MHz)': (3000, None, 200, 60, (2500, 4500)), #done
    'Memory Clock (in MHz)': (4000, None, 250, 200, (4000, 12000)), #done
    'Effective Speed (in Gbps)': (30, None, 500, 15, (20, 40)), #done
    'Memory Size (in GB)': (24, None, 800, 30, (24, 300)), #done
    'Memory Bus (in bit)': (512, None, 1000, 45, (512,9000)), #done
    'Bandwidth (in GB/s)': (600, None, 700, 15, (600, 7000)), #done
    'Shading Units': (5000, None, 750, 30, (5000, 21000)), #done
    'TMUs': (550, None, 550, 8, (550, 1000)), #done
    'ROPs': (150, None, 600, 25, (150, 250)), #done
    'SM Count': (150, None, 400, None, (140, 250)), #done
    'Tensor Cores': (750, None, 70, 10, (750, 1500)), #done
    'RT Cores': (160, None, 75, 15, (160, 500)), #done
    'L1 Cache (in KB)': (260, None, 700, 30, (260, 500)), #done
    'L2 Cache (in MB)': (10, None, 1000, 23, (10, 250)), #done
    'Pixel Rate (in GPixel/s)': (150, None, 800, 18, (150, 700)), #done
    'Texture Rate (in Gtexel/s)': (650, None, 800, 6, (650, 2000)), #done
    'FP16 (half, in TFLOPS)': (60, None, 400, 10, (60, 700)), #done
    'FP32 (float, in TFLOPS)': (30, None, 800, 4, (30, 60)), #done
    'FP64 (double, in TFLOPS)': (4, None, 500, 10, (4,85)), #done
    'TDP (in Watts)': (500, None, 450, 7, (450, 1100)), #done
}

# Call the function to plot histograms with custom limits
plot_histograms_with_custom_limits(gpu_data, columns_with_limits)

# Plot categorical bar plots
plot_categorical_barplots(gpu_data, gpu_categorical_columns)
```

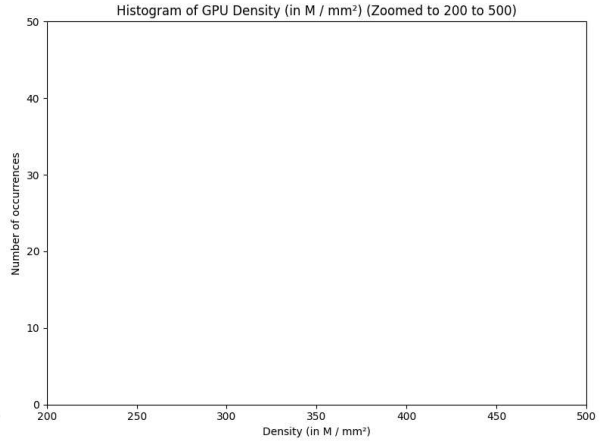


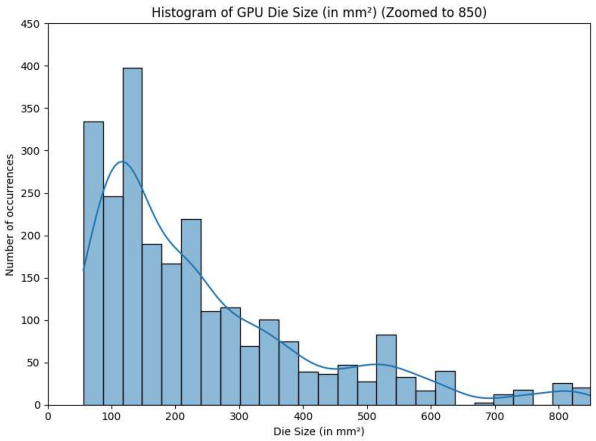


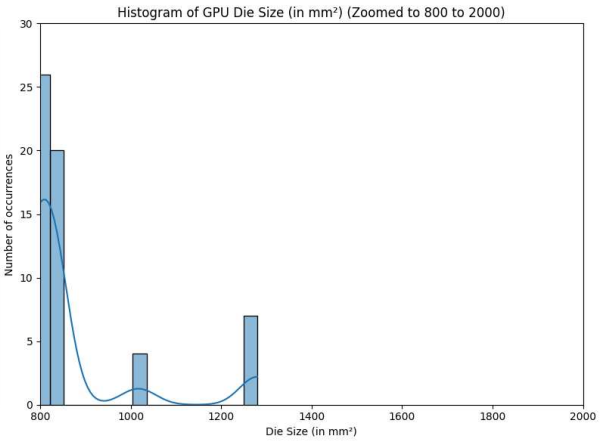

Histogram of GPU Transistors (in millions) (Zoomed to 5000)
Number of occurrences
Transistors (in millions)

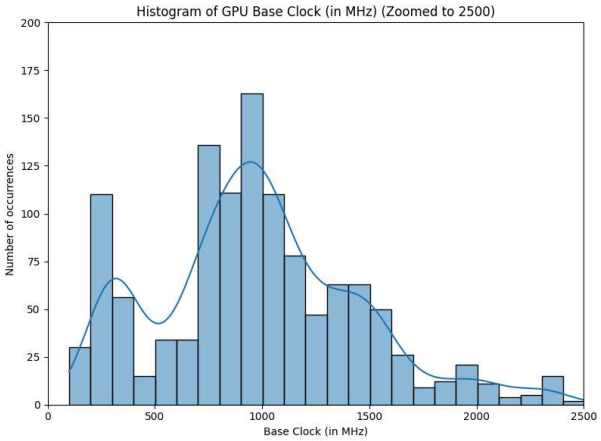

Histogram of GPU Transistors (in millions) (Zoomed to 5000 to 50000)
Number of occurrences
Transistors (in millions)

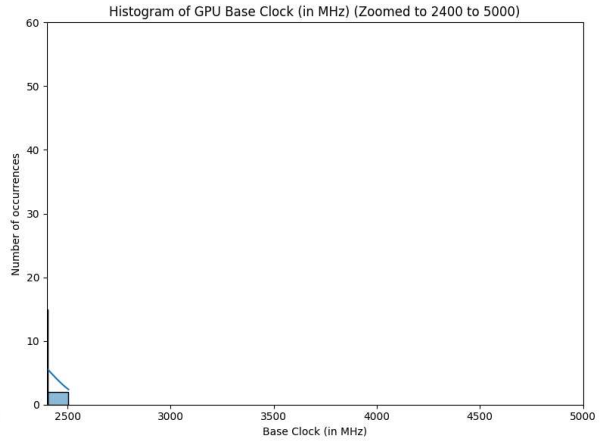

Histogram of GPU Density (in M / mm²) (Zoomed to 200)
Number of occurrences
Density (in M / mm²)

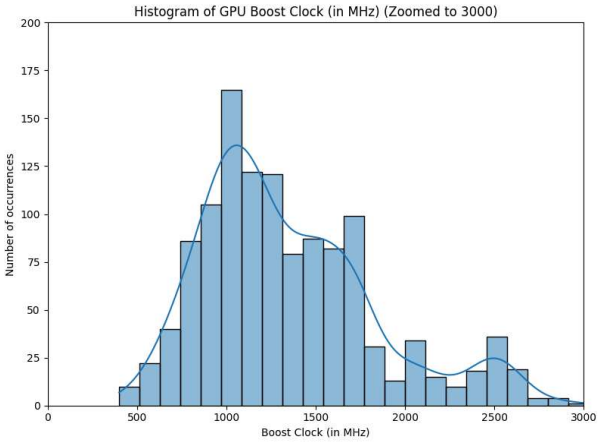

Histogram of GPU Density (in M / mm²) (Zoomed to 200 to 500)
Number of occurrences
Density (in M / mm²)

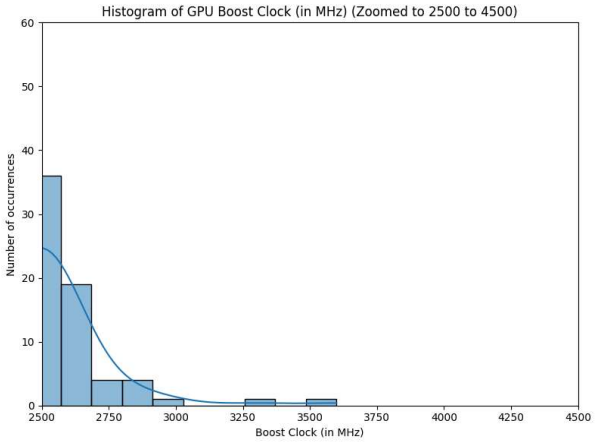

Histogram of GPU Die Size (in mm²) (Zoomed to 850)
Number of occurrences
Die Size (in mm²)

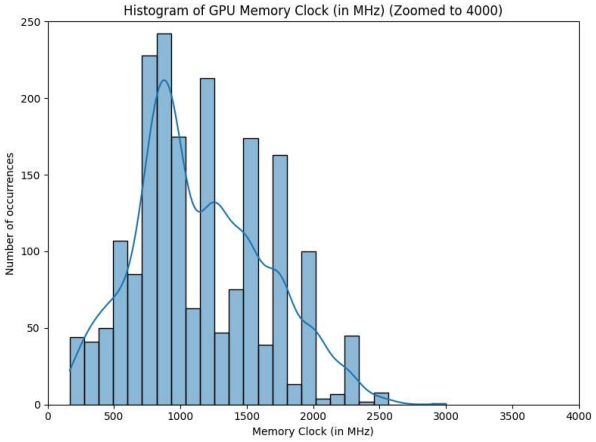

Histogram of GPU Die Size (in mm²) (Zoomed to 800 to 2000)
Number of occurrences
Die Size (in mm²)

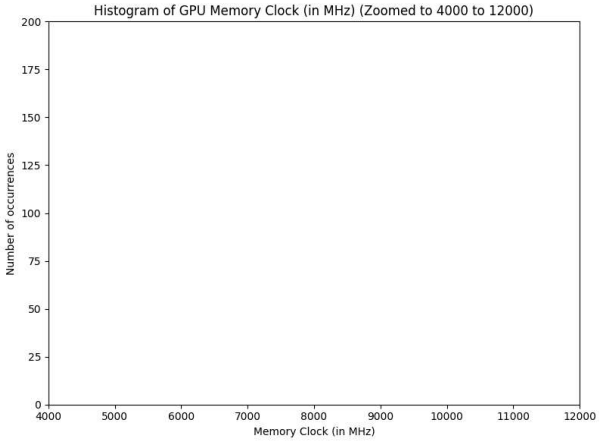

Histogram of GPU Base Clock (in MHz) (Zoomed to 2500)
Number of occurrences
Base Clock (in MHz)

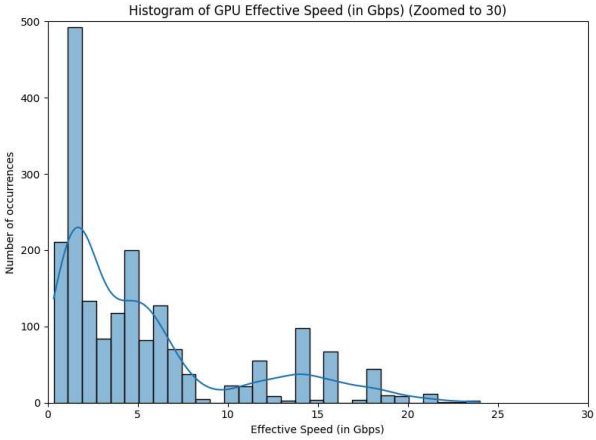

Histogram of GPU Base Clock (in MHz) (Zoomed to 2400 to 5000)
Number of occurrences
Base Clock (in MHz)

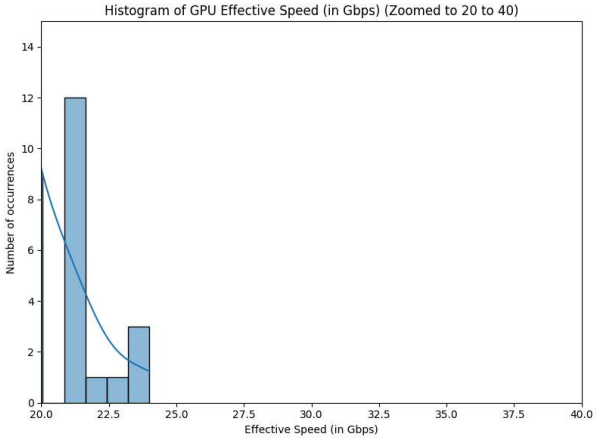

Histogram of GPU Boost Clock (in MHz) (Zoomed to 3000)
Number of occurrences
Boost Clock (in MHz)

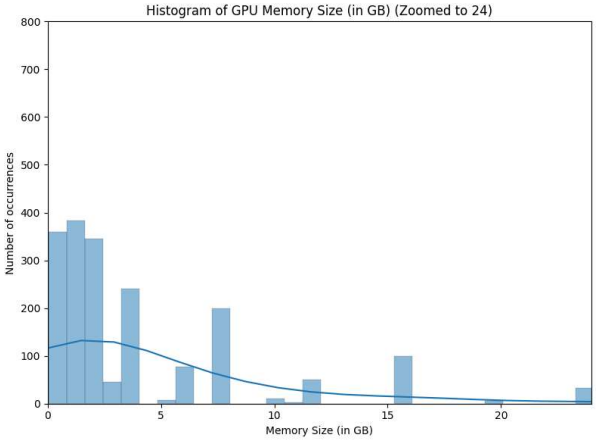

Histogram of GPU Boost Clock (in MHz) (Zoomed to 2500 to 4500)
Number of occurrences
Boost Clock (in MHz)

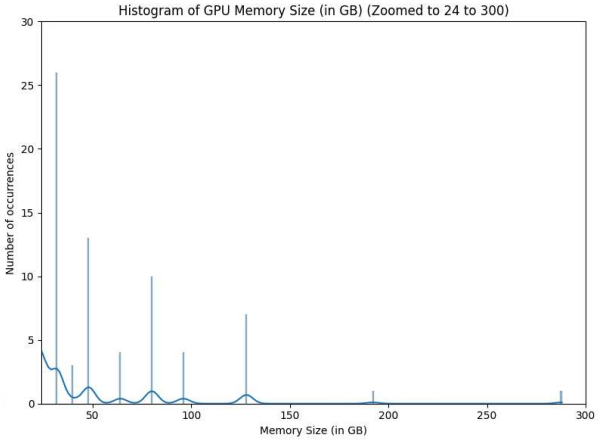

Histogram of GPU Memory Clock (in MHz) (Zoomed to 4000)
Number of occurrences
Memory Clock (in MHz)

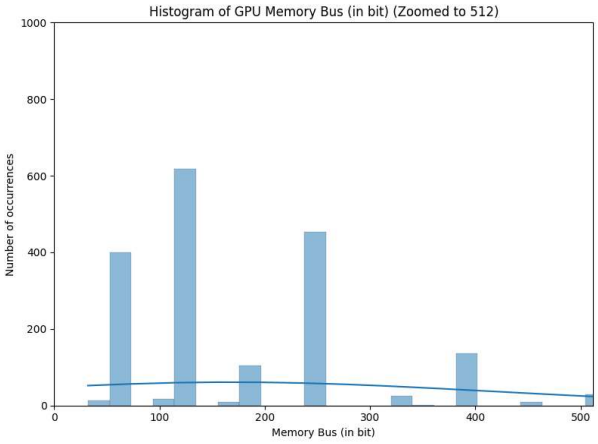

Histogram of GPU Memory Clock (in MHz) (Zoomed to 4000 to 12000)
Number of occurrences
Memory Clock (in MHz)

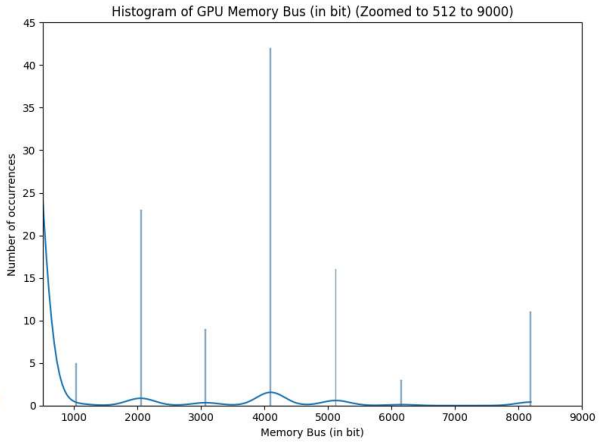

Histogram of GPU Effective Speed (in Gbps) (Zoomed to 30)
Number of occurrences
Effective Speed (in Gbps)

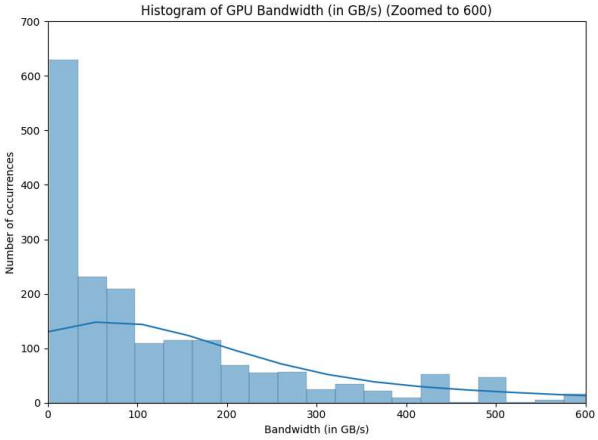

Histogram of GPU Effective Speed (in Gbps) (Zoomed to 20 to 40)
Number of occurrences
Effective Speed (in Gbps)

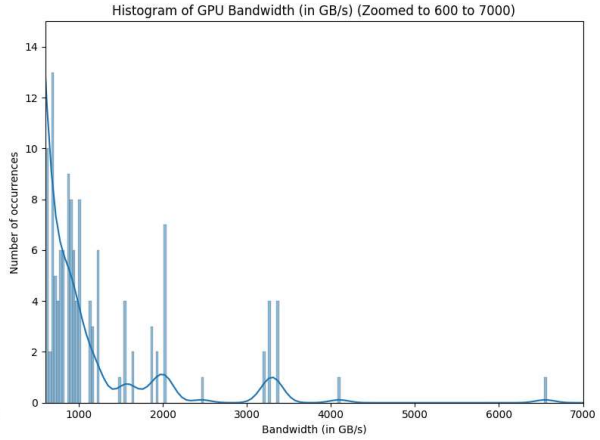

Histogram of GPU Memory Size (in GB) (Zoomed to 24)
Number of occurrences
Memory Size (in GB)

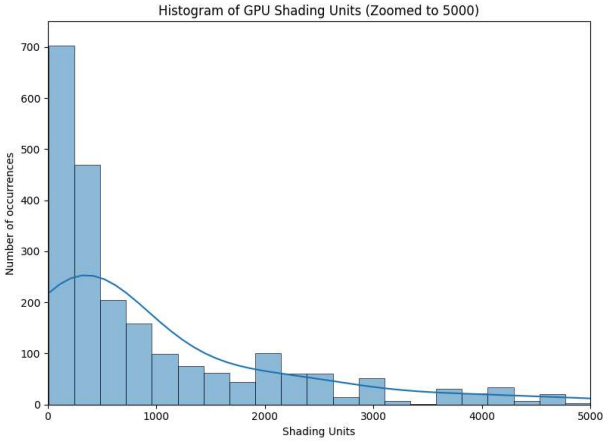

Histogram of GPU Memory Size (in GB) (Zoomed to 24 to 300)
Number of occurrences
Memory Size (in GB)

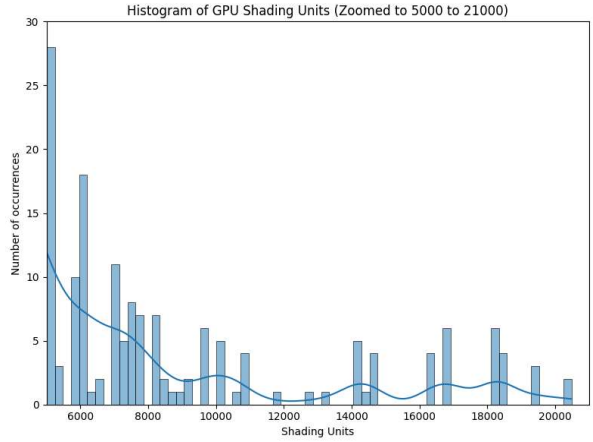

Histogram of GPU Memory Bus (in bit) (Zoomed to 512)
Number of occurrences
Memory Bus (in bit)

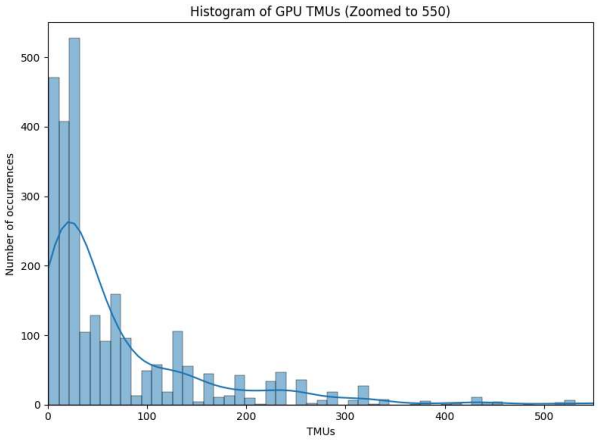

Histogram of GPU Memory Bus (in bit) (Zoomed to 512 to 9000)
Number of occurrences
Memory Bus (in bit)

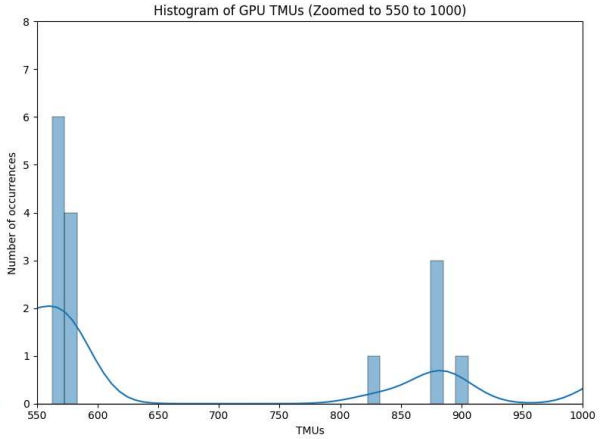

Histogram of GPU Bandwidth (in GB/s) (Zoomed to 600)
Number of occurrences
Bandwidth (in GB/s)

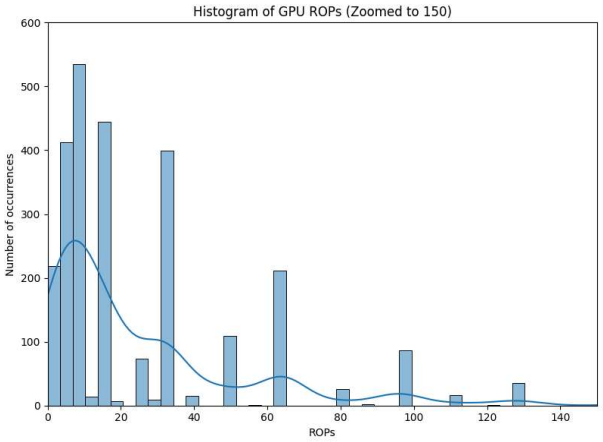

Histogram of GPU Bandwidth (in GB/s) (Zoomed to 600 to 7000)
Number of occurrences
Bandwidth (in GB/s)

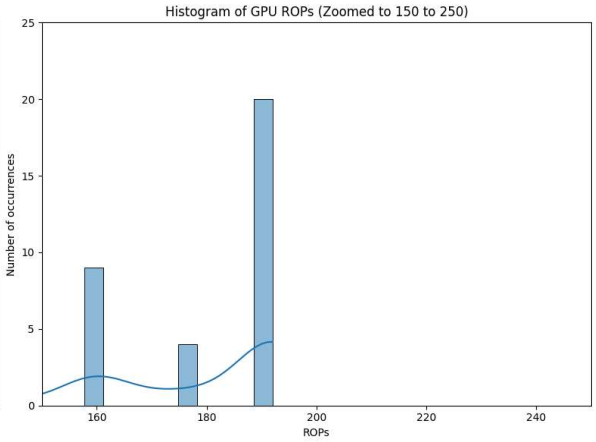

Histogram of GPU Shading Units (Zoomed to 5000)
Number of occurrences
Shading Units

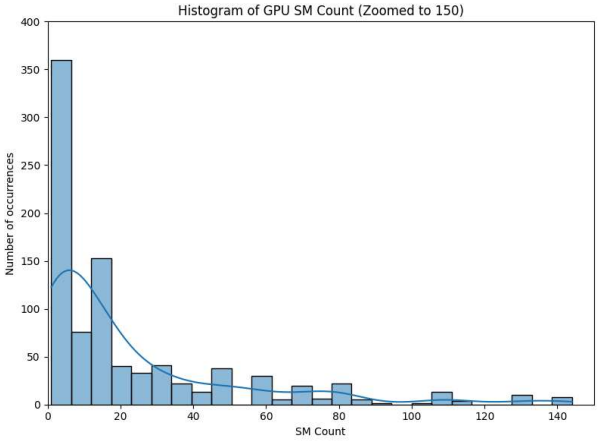

Histogram of GPU Shading Units (Zoomed to 5000 to 21000)
Number of occurrences
Shading Units

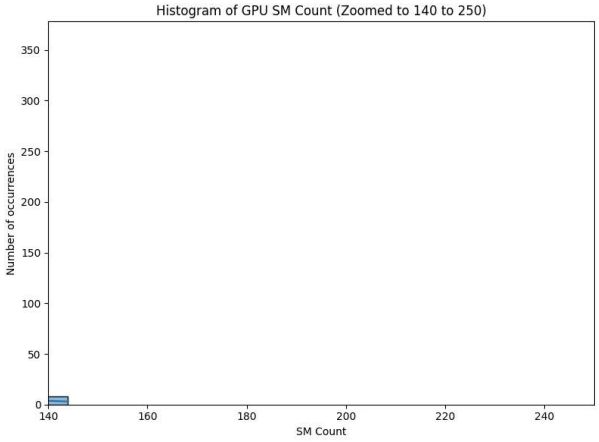

Histogram of GPU TMUs (Zoomed to 550)
Number of occurrences
TMUs

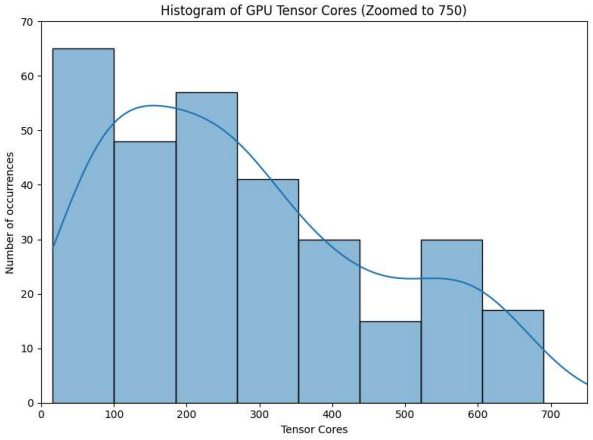

Histogram of GPU TMUs (Zoomed to 550 to 1000)
Number of occurrences
TMUs

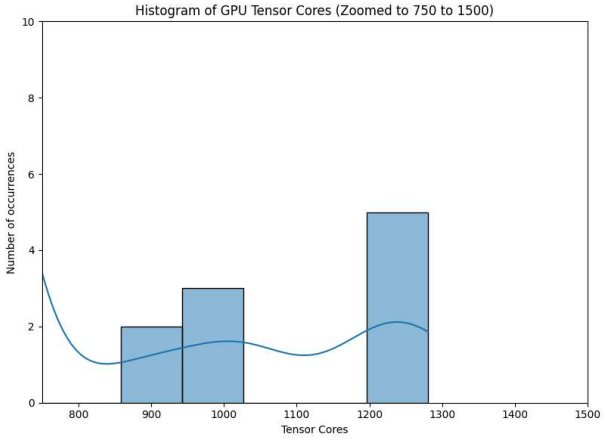

Histogram of GPU ROPs (Zoomed to 150)
Number of occurrences
ROPs

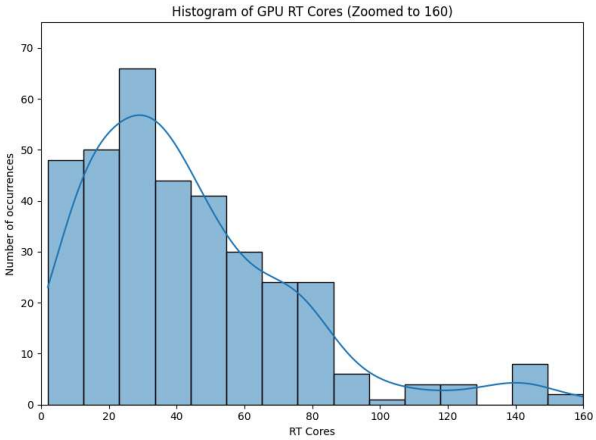

Histogram of GPU ROPs (Zoomed to 150 to 250)
Number of occurrences
ROPs

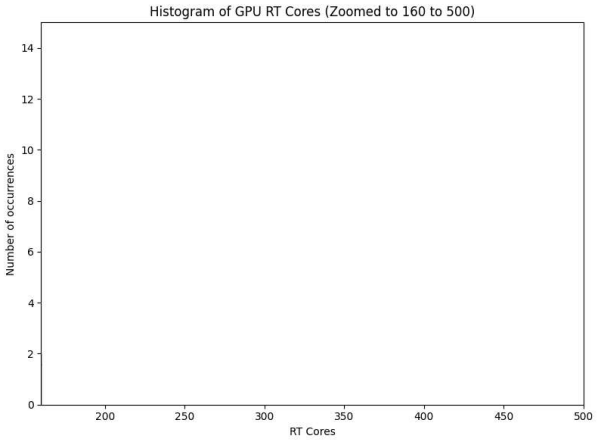

Histogram of GPU SM Count (Zoomed to 150)
Number of occurrences
SM Count

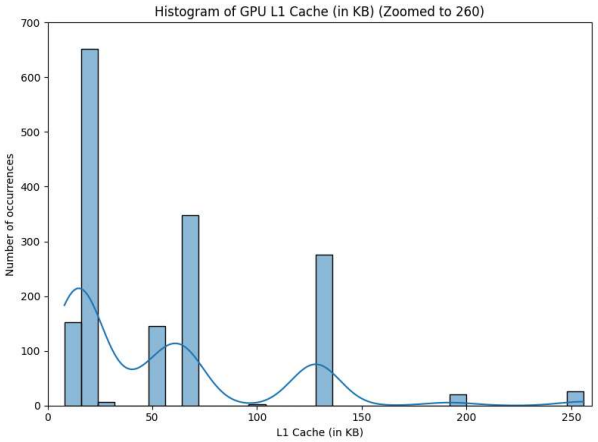

Histogram of GPU SM Count (Zoomed to 140 to 250)
Number of occurrences
SM Count

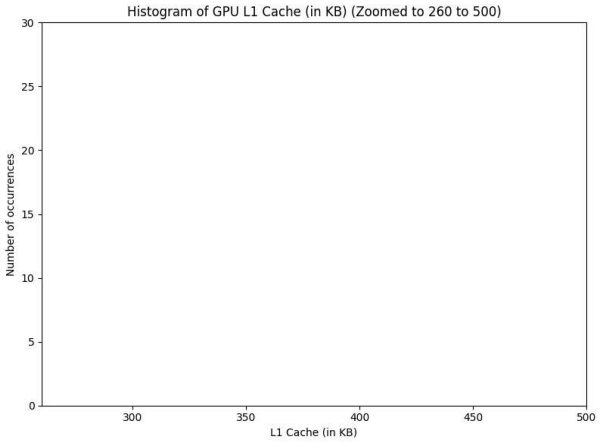

Histogram of GPU Tensor Cores (Zoomed to 750)
Number of occurrences
Tensor Cores

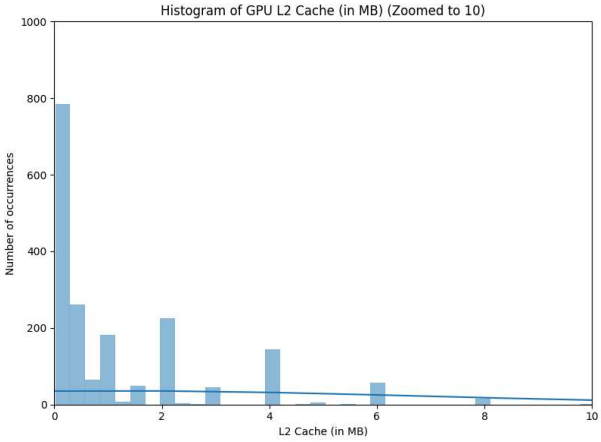

Histogram of GPU Tensor Cores (Zoomed to 750 to 1500)
Number of occurrences
Tensor Cores

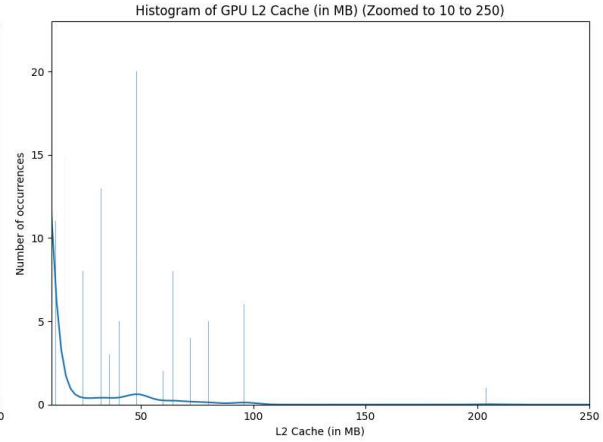

Histogram of GPU RT Cores (Zoomed to 160)
Number of occurrences
RT Cores

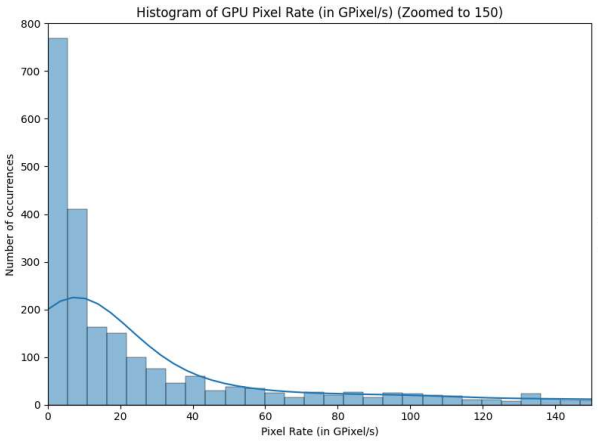

Histogram of GPU RT Cores (Zoomed to 160 to 500)
Number of occurrences
RT Cores

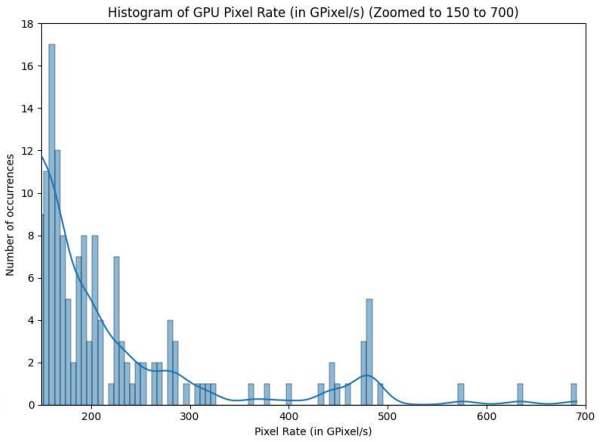

Histogram of GPU L1 Cache (in KB) (Zoomed to 260)
Number of occurrences
L1 Cache (in KB)

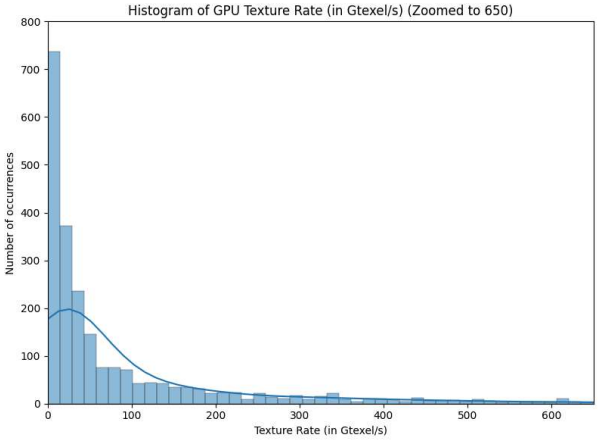

Histogram of GPU L1 Cache (in KB) (Zoomed to 260 to 500)
Number of occurrences
L1 Cache (in KB)

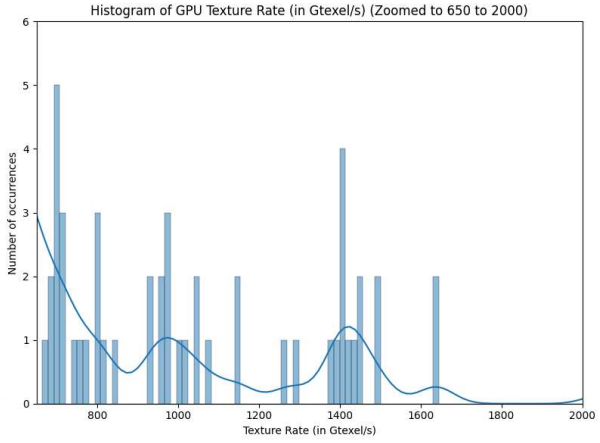

Histogram of GPU L2 Cache (in MB) (Zoomed to 10)
Number of occurrences
L2 Cache (in MB)

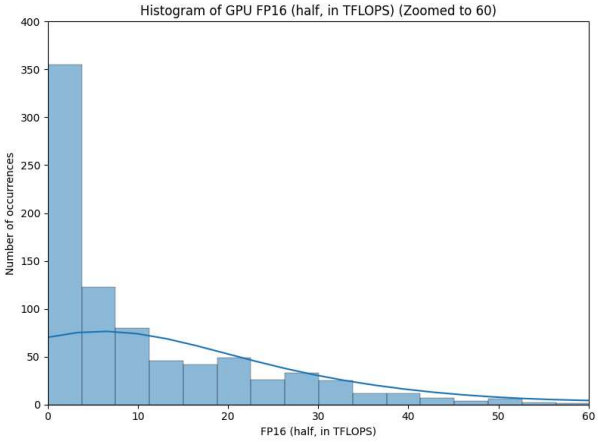

Histogram of GPU L2 Cache (in MB) (Zoomed to 10 to 250)
Number of occurrences
L2 Cache (in MB)

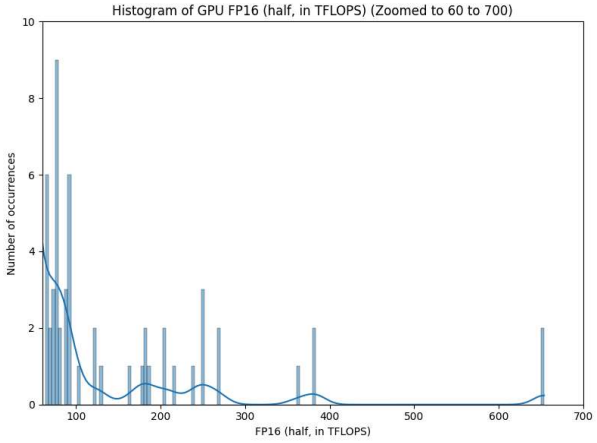

Histogram of GPU Pixel Rate (in GPixel/s) (Zoomed to 150)
Number of occurrences
Pixel Rate (in GPixel/s)

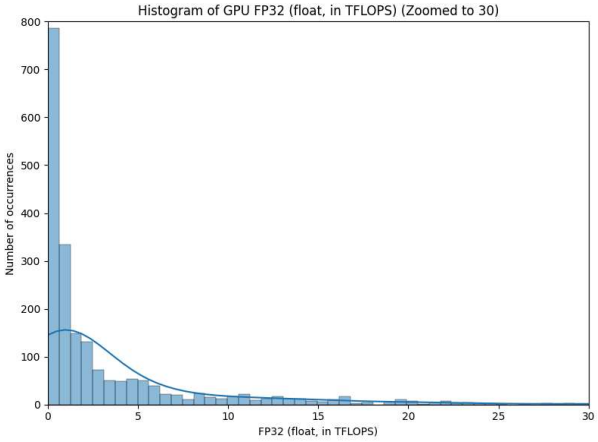

Histogram of GPU Pixel Rate (in GPixel/s) (Zoomed to 150 to 700)
Number of occurrences
Pixel Rate (in GPixel/s)

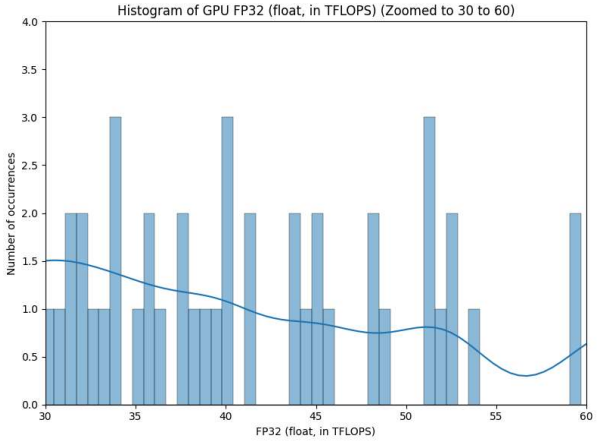

Histogram of GPU Texture Rate (in Gtexel/s) (Zoomed to 650)
Number of occurrences
Texture Rate (in Gtexel/s)

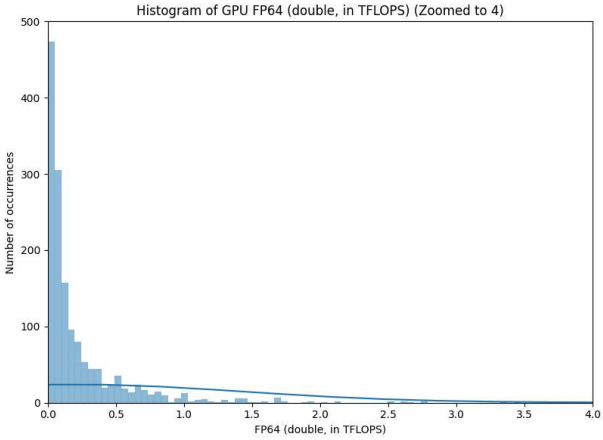

Histogram of GPU Texture Rate (in Gtexel/s) (Zoomed to 650 to 2000)
Number of occurrences
Texture Rate (in Gtexel/s)

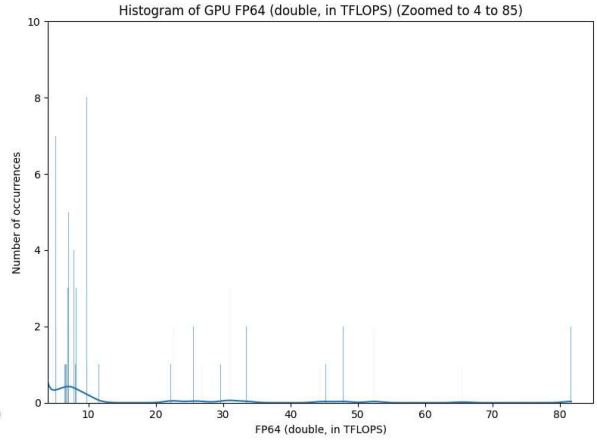

Histogram of GPU FP16 (half, in TFLOPS) (Zoomed to 60)
Number of occurrences
FP16 (half, in TFLOPS)

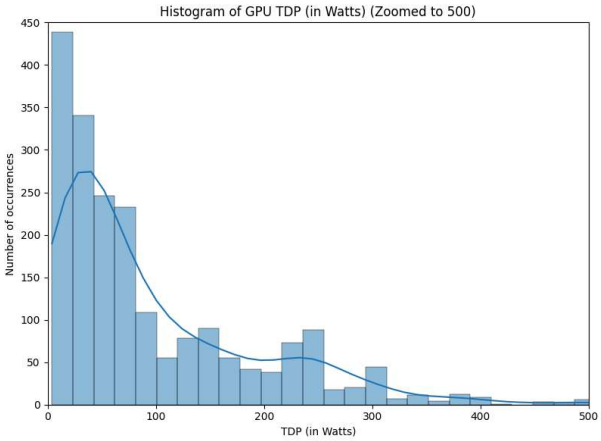

Histogram of GPU FP16 (half, in TFLOPS) (Zoomed to 60 to 700)
Number of occurrences
FP16 (half, in TFLOPS)

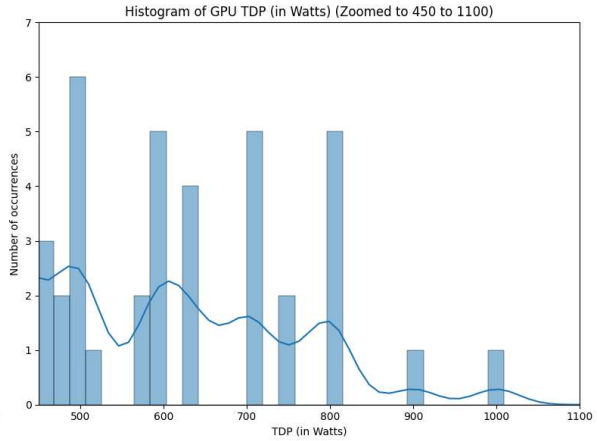

Histogram of GPU FP32 (float, in TFLOPS) (Zoomed to 30)
Number of occurrences
FP32 (float, in TFLOPS)

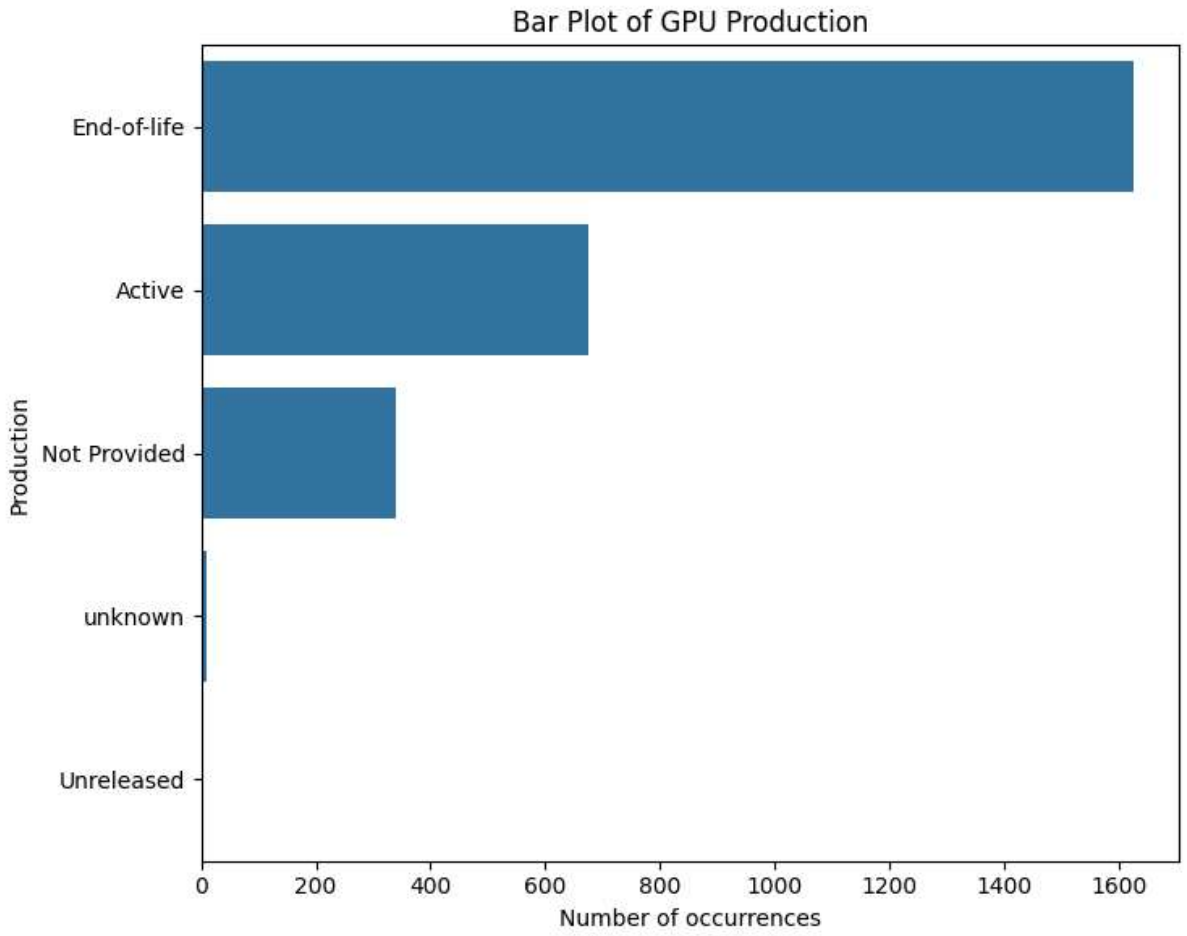

Histogram of GPU FP32 (float, in TFLOPS) (Zoomed to 30 to 60)
Number of occurrences
FP32 (float, in TFLOPS)


Histogram of GPU FP64 (double, in TFLOPS) (Zoomed to 4)
Number of occurrences
FP64 (double, in TFLOPS)


Histogram of GPU FP64 (double, in TFLOPS) (Zoomed to 4 to 85)
Number of occurrences
FP64 (double, in TFLOPS)


Histogram of GPU TDP (in Watts) (Zoomed to 500)
Number of occurrences
TDP (in Watts)

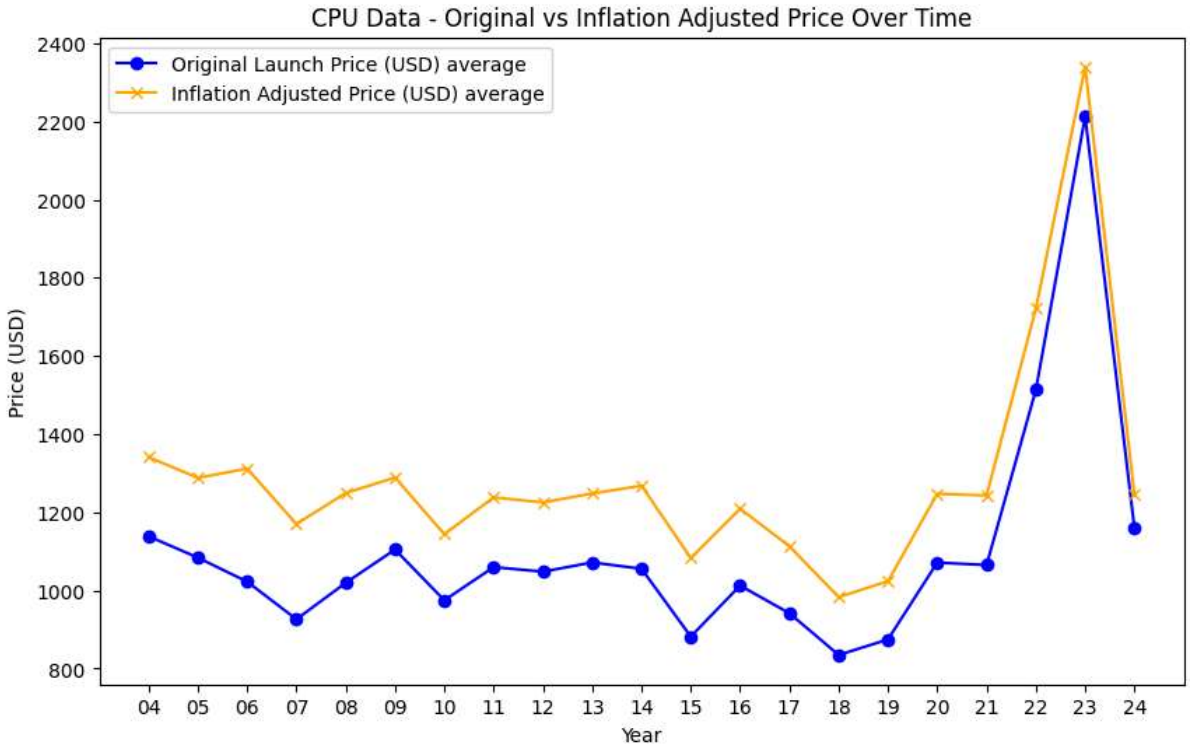

Histogram of GPU TDP (in Watts) (Zoomed to 450 to 1100)
Number of occurrences
TDP (in Watts)



CPU Launch Price and Inflation Adjusted Launch Price Over Time

In [5]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.ensemble import RandomForestRegressor

# Ensure 'Processor - Release Date' is in datetime format
cpu_data['Processor - Release Date'] = pd.to_datetime(cpu_data['Processor - Releas

# Convert 'Processor - Launch Price (in USD)' and 'Processor - Launch Price Inflat
cpu_data['Processor - Launch Price  (in USD)'] = pd.to_numeric(cpu_data['Processor
cpu_data['Processor - Launch Price Inflation Adjusted for 2024 (in USD)'] = pd.to_

# Fill NaN values with the mean of the column
cpu_data['Processor - Launch Price  (in USD)'] = cpu_data['Processor - Launch Pric
cpu_data['Processor - Launch Price Inflation Adjusted for 2024 (in USD)'] = cpu_da

# Extract the year from the release date
cpu_data['Release Year'] = cpu_data['Processor - Release Date'].dt.year

# Group by the year and calculate the mean values for launch price and inflation-a
cpu_yearly_avg = cpu_data.groupby('Release Year').agg({
    'Processor - Launch Price  (in USD)': 'mean',
    'Processor - Launch Price Inflation Adjusted for 2024 (in USD)': 'mean'
```

```
}).reset_index()

# Reindex the dataframe to ensure we have values for all years between 2004 and 20
cpu_yearly_avg = cpu_yearly_avg.set_index('Release Year').reindex(range(2004, 2025

# Plot the averaged data with adjusted x-ticks
def plot_original_vs_inflation_adjusted_price(data):
    plt.figure(figsize=(10, 6))

    # Plot both the original launch price and inflation-adjusted price
    plt.plot(data['Release Year'], data['Processor - Launch Price  (in USD)'], lab
    plt.plot(data['Release Year'], data['Processor - Launch Price Inflation Adjust

    plt.title('CPU Data - Original vs Inflation Adjusted Price Over Time')
    plt.xlabel('Year')
    plt.ylabel('Price (USD)')

    # Set x-ticks to only show the last two digits of the year
    years = np.arange(2004, 2025, 1)  # Full years from 2004 to 2024
    plt.xticks(years, [str(year)[-2:] for year in years])  # Extract the last two

    plt.legend()
    plt.show()

# Call the function to plot
plot_original_vs_inflation_adjusted_price(cpu_yearly_avg)
```

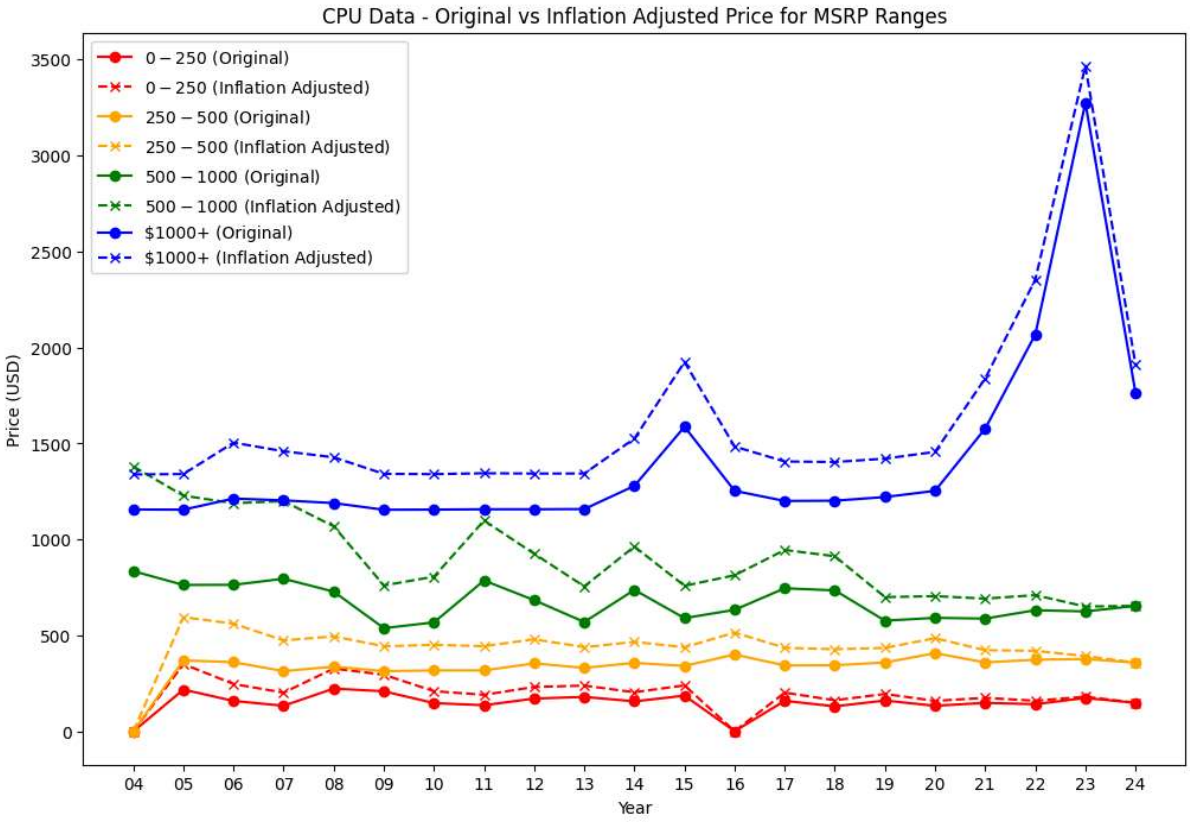


In [6]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Define values to drop in the 'Processor - Release Date' column
```

```
values_to_drop = ["Never Released", "Unknown", "Not Provided"]

# Filter out rows where 'Processor - Release Date' contains these values
cpu_data = cpu_data[~cpu_data['Processor - Release Date'].isin(values_to_drop)].co

# Ensure 'Processor - Release Date' is in datetime format
cpu_data.loc[:, 'Processor - Release Date'] = pd.to_datetime(cpu_data['Processor -

# Convert 'Processor - Launch Price (in USD)' and 'Processor - Launch Price Inflat
cpu_data.loc[:, 'Processor - Launch Price  (in USD)'] = pd.to_numeric(cpu_data['Pr
cpu_data.loc[:, 'Processor - Launch Price Inflation Adjusted for 2024 (in USD)'] =

# Fill NaN values with the mean of the column
cpu_data.loc[:, 'Processor - Launch Price  (in USD)'] = cpu_data['Processor - Laun
cpu_data.loc[:, 'Processor - Launch Price Inflation Adjusted for 2024 (in USD)'] =

# Function to filter data by MSRP ranges and return the yearly average for plottin
def get_cpu_yearly_avg(data, lower_limit=None, upper_limit=None):
    if lower_limit is not None and upper_limit is not None:
        filtered_cpu_data = data[(data['Processor - Launch Price  (in USD)'] >= lo
    else:
        filtered_cpu_data = data[data['Processor - Launch Price  (in USD)'] >= 100

    filtered_cpu_data.loc[:, 'Release Year'] = filtered_cpu_data['Processor - Relea
    cpu_yearly_avg = filtered_cpu_data.groupby('Release Year').agg({
        'Processor - Launch Price  (in USD)': 'mean',
        'Processor - Launch Price Inflation Adjusted for 2024 (in USD)': 'mean'
    }).reset_index()

    cpu_yearly_avg = cpu_yearly_avg.set_index('Release Year').reindex(range(2004,
    return cpu_yearly_avg

# Get data for different price ranges
cpu_yearly_avg_0_250 = get_cpu_yearly_avg(cpu_data, 0, 250)
cpu_yearly_avg_250_500 = get_cpu_yearly_avg(cpu_data, 250, 500)
cpu_yearly_avg_500_1000 = get_cpu_yearly_avg(cpu_data, 500, 1000)
cpu_yearly_avg_1000_above = get_cpu_yearly_avg(cpu_data, 1000, None)

# Plot the data for the different MSRP ranges
plt.figure(figsize=(12, 8))

# Plot for MSRP between 0 and 250
plt.plot(cpu_yearly_avg_0_250['Release Year'], cpu_yearly_avg_0_250['Processor - L
plt.plot(cpu_yearly_avg_0_250['Release Year'], cpu_yearly_avg_0_250['Processor - L

# Plot for MSRP between 250 and 500
plt.plot(cpu_yearly_avg_250_500['Release Year'], cpu_yearly_avg_250_500['Processor
plt.plot(cpu_yearly_avg_250_500['Release Year'], cpu_yearly_avg_250_500['Processor

# Plot for MSRP between 500 and 1000
plt.plot(cpu_yearly_avg_500_1000['Release Year'], cpu_yearly_avg_500_1000['Process
plt.plot(cpu_yearly_avg_500_1000['Release Year'], cpu_yearly_avg_500_1000['Process

# Plot for MSRP 1000 and above
plt.plot(cpu_yearly_avg_1000_above['Release Year'], cpu_yearly_avg_1000_above['Pro
plt.plot(cpu_yearly_avg_1000_above['Release Year'], cpu_yearly_avg_1000_above['Pro
```

```
# Title and labels
plt.title('CPU Data - Original vs Inflation Adjusted Price for MSRP Ranges')
plt.xlabel('Year')
plt.ylabel('Price (USD)')
years = np.arange(2004, 2025, 1)
plt.xticks(years, [str(year)[-2:] for year in years])
plt.legend()
plt.show()
```

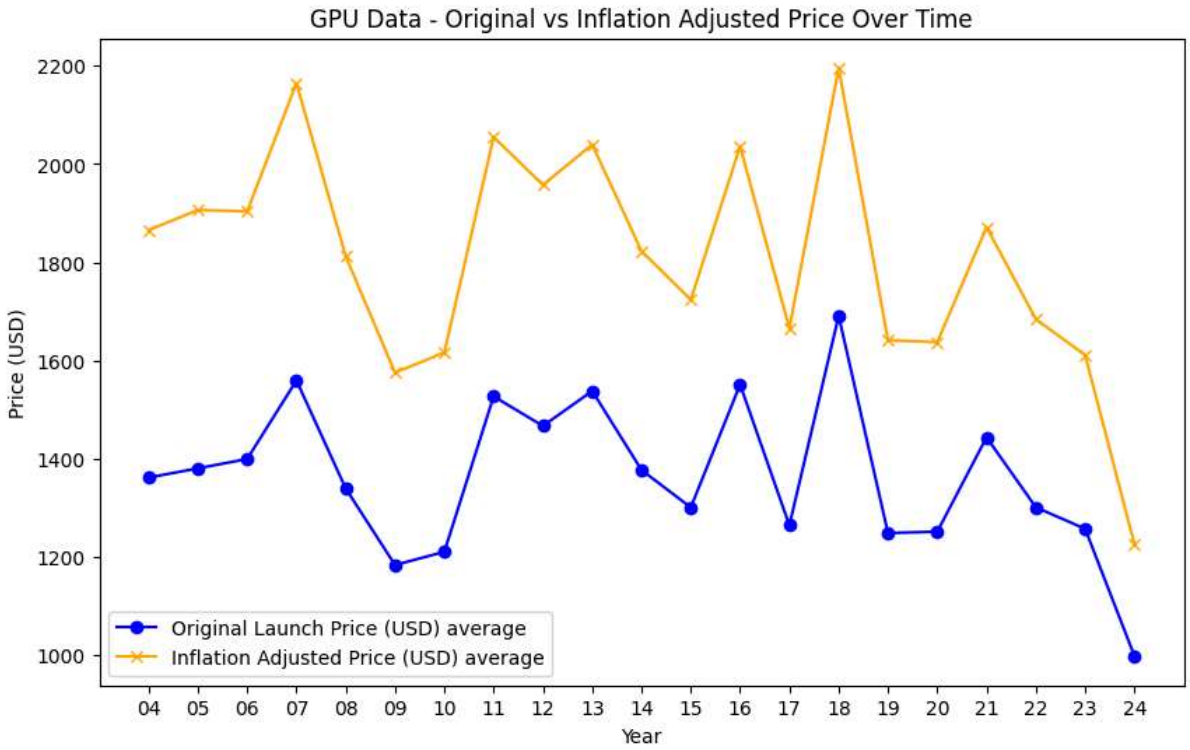


GPU Price vs. Performance Over Time

In [7]:

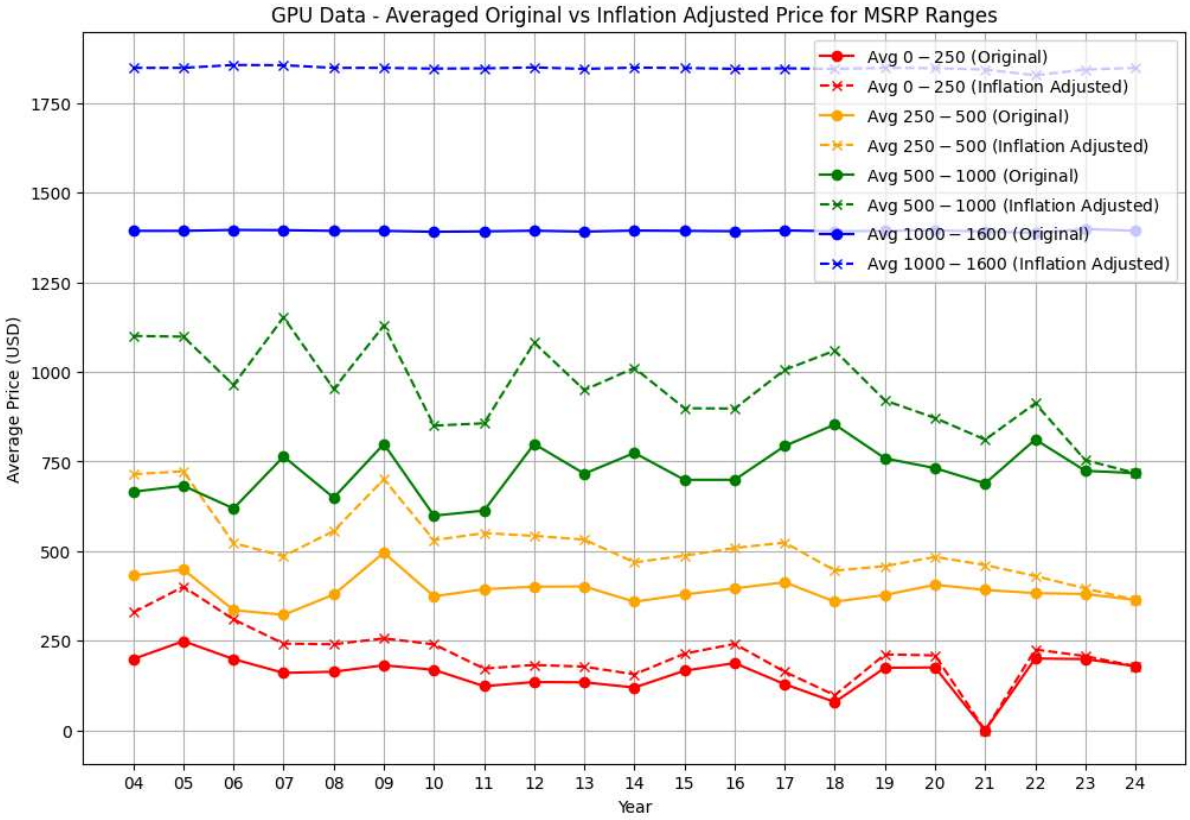
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Ensure 'Release Date' is in datetime format
gpu_data['Release Date'] = pd.to_datetime(gpu_data['Release Date'], errors='coerce

# Convert 'GPU - Launch Price (in USD) ' and 'GPU Launch Price adjusted for inflat
gpu_data['GPU - Launch Price (in USD)'] = pd.to_numeric(gpu_data['GPU - Launch Pri
gpu_data['GPU Launch Price adjusted for inflation in 2024 (in USD)'] = pd.to_numer

# Fill NaN values with the mean of the column
gpu_data['GPU - Launch Price (in USD)'] = gpu_data['GPU - Launch Price (in USD)'].
gpu_data['GPU Launch Price adjusted for inflation in 2024 (in USD)'] = gpu_data['G

# Extract the year from the release date
gpu_data['Release Year'] = gpu_data['Release Date'].dt.year
```

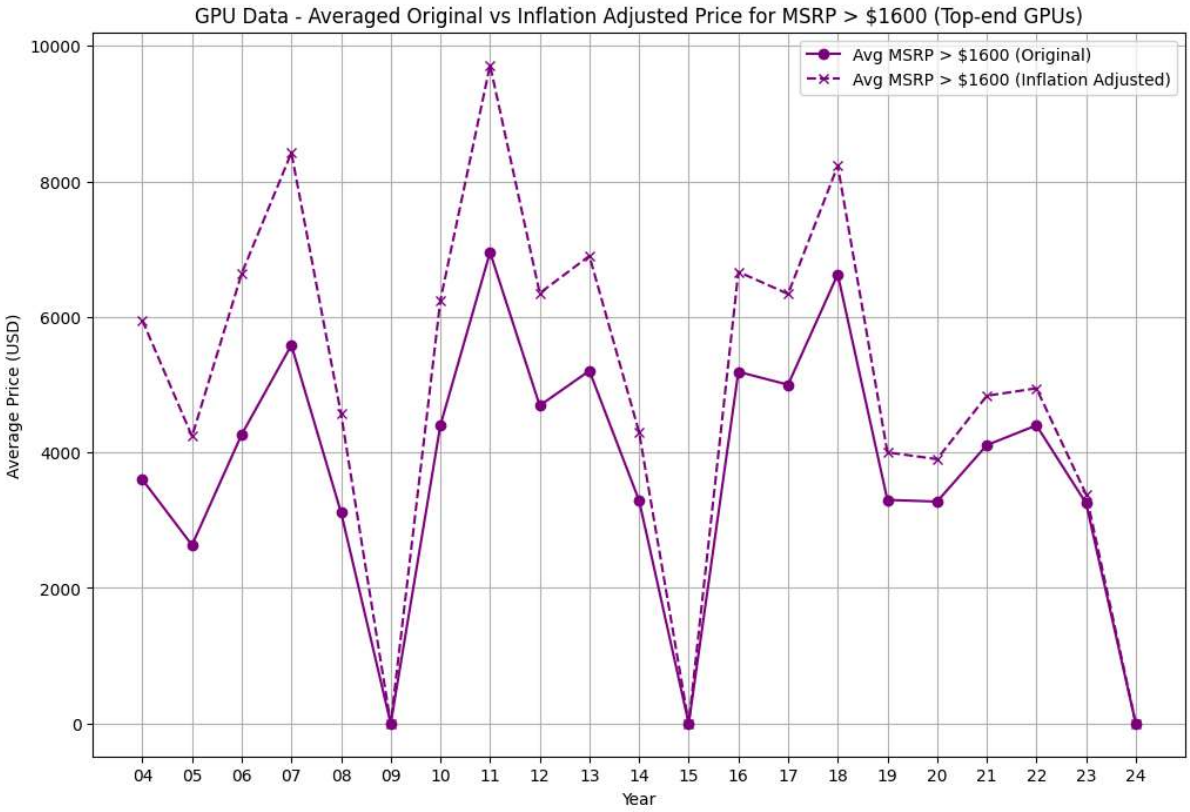
```
# Group by the year and calculate the mean values for launch price and inflation-a
gpu_yearly_avg = gpu_data.groupby('Release Year').agg({
    'GPU - Launch Price (in USD)': 'mean',
    'GPU Launch Price adjusted for inflation in 2024 (in USD)': 'mean'
}).reset_index()

# Reindex the dataframe to ensure we have values for all years between 2004 and 20
gpu_yearly_avg = gpu_yearly_avg.set_index('Release Year').reindex(range(2004, 2025

# Plot the averaged data with adjusted x-ticks
def plot_original_vs_inflation_adjusted_price_gpu(data):
    plt.figure(figsize=(10, 6))

    # Plot both the original launch price and inflation-adjusted price
    plt.plot(data['Release Year'], data['GPU - Launch Price (in USD)'], label='Ori
    plt.plot(data['Release Year'], data['GPU Launch Price adjusted for inflation i

    plt.title('GPU Data - Original vs Inflation Adjusted Price Over Time')
    plt.xlabel('Year')
    plt.ylabel('Price (USD)')

    # Set x-ticks to only show the last two digits of the year
    years = np.arange(2004, 2025, 1)  # Full years from 2004 to 2024
    plt.xticks(years, [str(year)[-2:] for year in years])  # Extract the last two

    plt.legend()
    plt.show()

# Call the function to plot
plot_original_vs_inflation_adjusted_price_gpu(gpu_yearly_avg)
```



In [8]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Define the values to drop in the 'Release Date' column
values_to_drop = ["Never Released", "Unknown", "Not Provided"]

# Filter out rows where 'Release Date' contains these values
gpu_data = gpu_data[~gpu_data['Release Date'].isin(values_to_drop)]

# Ensure 'Release Date' is in datetime format
gpu_data['Release Date'] = pd.to_datetime(gpu_data['Release Date'], errors='coerce

# Convert 'GPU - Launch Price (in USD)' and 'GPU Launch Price adjusted for inflati
gpu_data['GPU - Launch Price (in USD)'] = pd.to_numeric(gpu_data['GPU - Launch Pri
gpu_data['GPU Launch Price adjusted for inflation in 2024 (in USD)'] = pd.to_numer

# Fill NaN values with the mean of the column
gpu_data['GPU - Launch Price (in USD)'] = gpu_data['GPU - Launch Price (in USD)'].
gpu_data['GPU Launch Price adjusted for inflation in 2024 (in USD)'] = gpu_data['G

# Function to filter data by MSRP ranges (inclusive lower bound, exclusive upper b
def get_gpu_yearly_avg(data, lower_limit=None, upper_limit=None):
    if lower_limit is not None and upper_limit is not None:
        # Use inclusive lower limit and exclusive upper limit for true ranges
        filtered_gpu_data = data[(data['GPU - Launch Price (in USD)'] >= lower_lim
    elif lower_limit is not None:
        # For the top-end GPUs (e.g., 1600+), include the entire range above the l
        filtered_gpu_data = data[data['GPU - Launch Price (in USD)'] >= lower_limi

    filtered_gpu_data['Release Year'] = filtered_gpu_data['Release Date'].dt.year
    gpu_yearly_avg = filtered_gpu_data.groupby('Release Year').agg({
        'GPU - Launch Price (in USD)': 'mean',
        'GPU Launch Price adjusted for inflation in 2024 (in USD)': 'mean'
    }).reset_index()

    gpu_yearly_avg = gpu_yearly_avg.set_index('Release Year').reindex(range(2004,
    return gpu_yearly_avg

# Define true ranges for MSRP brackets
gpu_yearly_avg_0_250 = get_gpu_yearly_avg(gpu_data, 0, 250)    # $0 - $250
gpu_yearly_avg_250_500 = get_gpu_yearly_avg(gpu_data, 250, 500) # $250 - $500
gpu_yearly_avg_500_1000 = get_gpu_yearly_avg(gpu_data, 500, 1000) # $500 - $1000
gpu_yearly_avg_1000_1600 = get_gpu_yearly_avg(gpu_data, 1000, 1600) # $1000 - $160

# Plot the data with the specified ranges and emphasize averaging
plt.figure(figsize=(12, 8))

# Plot for MSRP between $0 and $250
plt.plot(gpu_yearly_avg_0_250['Release Year'], gpu_yearly_avg_0_250['GPU - Launch
plt.plot(gpu_yearly_avg_0_250['Release Year'], gpu_yearly_avg_0_250['GPU Launch Pr

# Plot for MSRP between $250 and $500
plt.plot(gpu_yearly_avg_250_500['Release Year'], gpu_yearly_avg_250_500['GPU - Lau
```

```
plt.plot(gpu_yearly_avg_250_500['Release Year'], gpu_yearly_avg_250_500['GPU Launc

# Plot for MSRP between $500 and $1000
plt.plot(gpu_yearly_avg_500_1000['Release Year'], gpu_yearly_avg_500_1000['GPU - I
plt.plot(gpu_yearly_avg_500_1000['Release Year'], gpu_yearly_avg_500_1000['GPU Lau

# Plot for MSRP between $1000 and $1600
plt.plot(gpu_yearly_avg_1000_1600['Release Year'], gpu_yearly_avg_1000_1600['GPU -
plt.plot(gpu_yearly_avg_1000_1600['Release Year'], gpu_yearly_avg_1000_1600['GPU I

# Title and labels for clarity
plt.title('GPU Data - Averaged Original vs Inflation Adjusted Price for MSRP Range
plt.xlabel('Year')
plt.ylabel('Average Price (USD)')
years = np.arange(2004, 2025, 1)
plt.xticks(years, [str(year)[-2:] for year in years])
plt.legend(loc='best')
plt.grid(True)
plt.show()

# Second version: Plot MSRP > $1600 (Top-end GPUs)
gpu_yearly_avg_top_end = get_gpu_yearly_avg(gpu_data, 1600, None)

# Plot the top-end GPUs in a separate plot
plt.figure(figsize=(12, 8))

# Plot for MSRP > $1600
plt.plot(gpu_yearly_avg_top_end['Release Year'], gpu_yearly_avg_top_end['GPU - Lau
plt.plot(gpu_yearly_avg_top_end['Release Year'], gpu_yearly_avg_top_end['GPU Launc

# Title and labels for clarity
plt.title('GPU Data - Averaged Original vs Inflation Adjusted Price for MSRP > $16
plt.xlabel('Year')
plt.ylabel('Average Price (USD)')
plt.xticks(years, [str(year)[-2:] for year in years])
plt.legend(loc='best')
plt.grid(True)
plt.show()
```



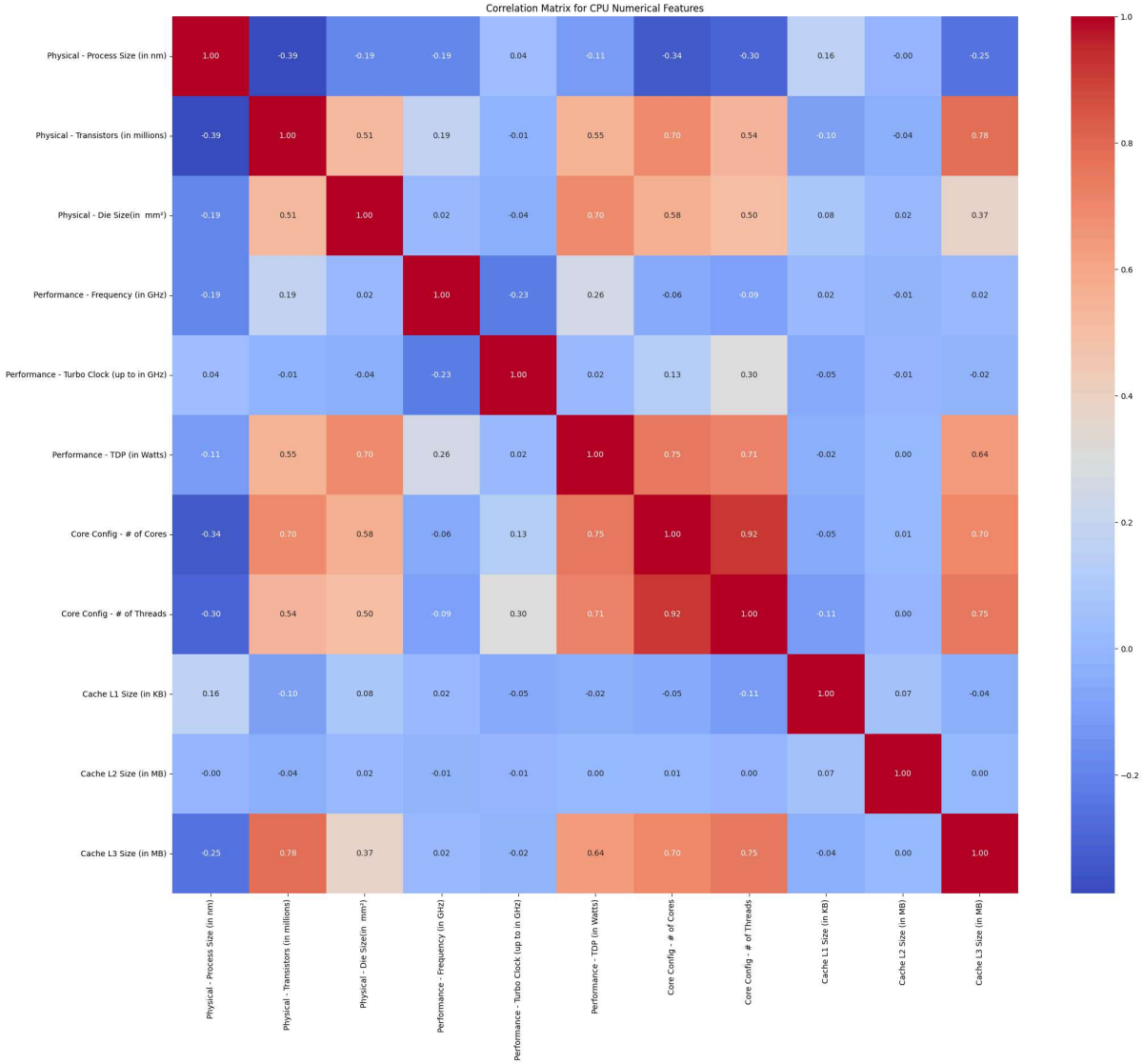


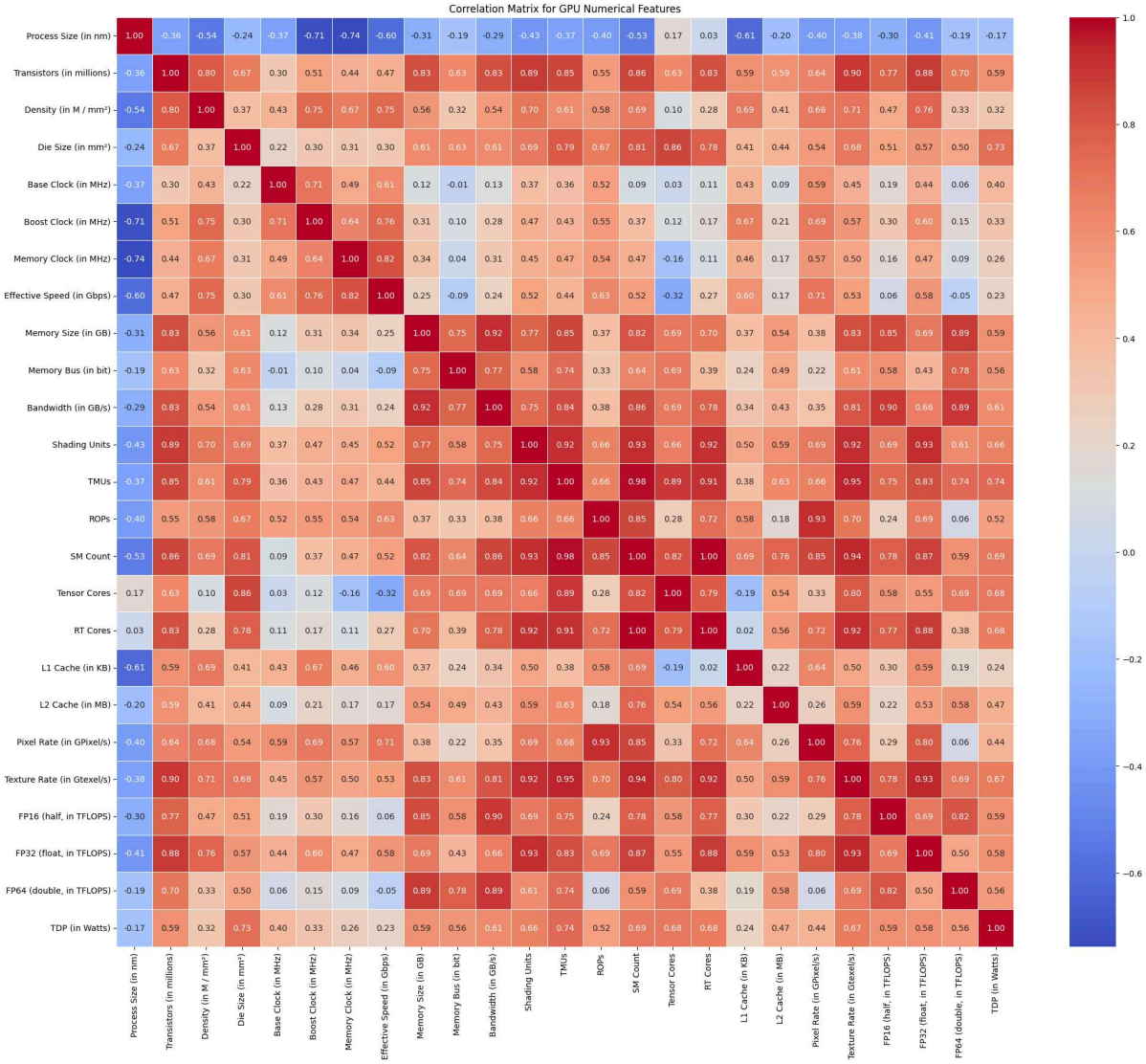
Correlation Matrix for CPU and GPU

In [9]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

```
# Load the CPU data
cpu_data = pd.read_excel('../src/cpu_data_cleaned.xlsx')

# Strip any leading or trailing spaces from column names
cpu_data.columns = cpu_data.columns.str.strip()

# List of numerical columns you're interested in
cpu_numerical_columns = [
    'Physical - Process Size (in nm)',
    'Physical - Transistors (in millions)',
    'Physical - Die Size(in  mm²)',
    'Performance - Frequency (in GHz)',
    'Performance - Turbo Clock (up to in GHz)',
    'Performance - TDP (in Watts)',
    'Core Config - # of Cores',
    'Core Config - # of Threads',
    'Cache L1 Size (in KB)',
    'Cache L2 Size (in MB)',
    'Cache L3 Size (in MB)'
]

# Convert the numerical columns to numeric, in case of any non-numeric data
cpu_data[cpu_numerical_columns] = cpu_data[cpu_numerical_columns].apply(pd.to_nume

# Generate the correlation matrix for the numerical columns
correlation_matrix = cpu_data[cpu_numerical_columns].corr()

# Plotting the correlation matrix using seaborn
plt.figure(figsize=(24, 20))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Correlation Matrix for CPU Numerical Features')
plt.show()


# Load the GPU data
gpu_data = pd.read_excel('../src/gpu_data_cleaned.xlsx')

# Strip any leading or trailing spaces from column names
gpu_data.columns = gpu_data.columns.str.strip()

# List of GPU numerical columns
gpu_numerical_columns = [
    "Process Size (in nm)",
    "Transistors (in millions)",
    "Density (in M / mm²)",
    "Die Size (in mm²)",
    "Base Clock (in MHz)",
    "Boost Clock (in MHz)",
    "Memory Clock (in MHz)",
    "Effective Speed (in Gbps)",
    "Memory Size (in GB)",
    "Memory Bus (in bit)",
    "Bandwidth (in GB/s)",
    "Shading Units",
    "TMUs",
```

```
    "ROPs",
    "SM Count",
    "Tensor Cores",
    "RT Cores",
    "L1 Cache (in KB)",
    "L2 Cache (in MB)",
    "Pixel Rate (in GPixel/s)",
    "Texture Rate (in Gtexel/s)",
    "FP16 (half, in TFLOPS)",
    "FP32 (float, in TFLOPS)",
    "FP64 (double, in TFLOPS)",
    "TDP (in Watts)"
]

# Convert the numerical columns to numeric, in case of any non-numeric data
gpu_data[gpu_numerical_columns] = gpu_data[gpu_numerical_columns].apply(pd.to_nume

# Generate the correlation matrix for the numerical columns
gpu_correlation_matrix = gpu_data[gpu_numerical_columns].corr()

# Plotting the correlation matrix using seaborn
plt.figure(figsize=(24, 20))
sns.heatmap(gpu_correlation_matrix, annot=True, cmap='coolwarm', fmt='.2f', linewi
            xticklabels=gpu_correlation_matrix.columns, yticklabels=gpu_correlatio

# Rotate the x-axis labels vertically
plt.xticks(rotation=90, ha='center')  # Rotate and align x-axis labels in vertical

plt.title('Correlation Matrix for GPU Numerical Features')
plt.show()
```


Correlation Matrix for CPU Numerical Features
Physical - Process Size (in nm)
Physical - Transistors (in millions)
Physical - Die Size(in mm²)
Performance - Frequency (in GHz)
Performance - Turbo Clock (up to in GHz)
Performance - TDP (in Watts)
Core Config - # of Cores
Core Config - # of Threads
Cache L1 Size (in KB)
Cache L2 Size (in MB)
Cache L3 Size (in MB)
1.00 -0.39 -0.19 -0.19 0.04 -0.11 -0.34 -0.30 0.16 -0.00 -0.25
-0.39 1.00 0.51 0.19 -0.01 0.55 0.70 0.54 -0.10 -0.04 0.78
-0.19 0.51 1.00 0.02 -0.04 0.70 0.58 0.50 0.08 0.02 0.37
-0.19 0.19 0.02 1.00 -0.23 0.26 -0.06 -0.09 0.02 -0.01 0.02
0.04 -0.01 -0.04 -0.23 1.00 0.02 0.13 0.30 -0.05 -0.01 -0.02
-0.11 0.55 0.70 0.26 0.02 1.00 0.75 0.71 -0.02 0.00 0.64
-0.34 0.70 0.58 -0.06 0.13 0.75 1.00 0.92 -0.05 0.01 0.70
-0.30 0.54 0.50 -0.09 0.30 0.71 0.92 1.00 -0.11 0.00 0.75
0.16 -0.10 0.08 0.02 -0.05 -0.02 -0.05 -0.11 1.00 0.07 -0.04
-0.00 -0.04 0.02 -0.01 -0.01 0.00 0.01 0.00 0.07 1.00 0.00
-0.25 0.78 0.37 0.02 -0.02 0.64 0.70 0.75 -0.04 0.00 1.00
1.0
0.8
0.6
0.4
0.2
0.0
−0.2



Exploratory Data Analysis of TechPowerUp's comprehensive data on CPU and GPU

The goal of this discussion is to explore general trends in the pricing, physical and performance characteristics of CPUs and GPUs over the years, and identify potential relationships between key features. This EDA will be used as a prerequisite step for predictive modeling, primarily focusing on predicting the launch price of these components.

---

1. Distribution of Features (Histograms)

CPU Data

- **Physical Features**:

  - The **process size** has seen a steady decrease over time, with the majority of CPUs in the data using process sizes between 10nm and 32nm. Larger process sizes (e.g., 130nm) are increasingly rare.
  - **Transistor count** shows a right-skewed distribution, with a small number of CPUs featuring extremely high transistor counts (>10,000 million). This reflects the cutting-edge designs of modern CPUs.
  - **Die size** distributions show that most CPUs are clustered below 500 mm^2, though there are a few outliers with significantly larger die sizes.
- **Performance Features**:
  - **Clock speeds (GHz)** show a bell-curve-like distribution with most CPUs falling between 2 to 4 GHz, reflecting the balance between power consumption and performance.
  - **TDP** values exhibit a highly skewed distribution, with the majority of CPUs consuming below 200 watts, though a few outliers reach above 400W.

GPU Data

- **Process size** distributions for GPUs mirror those of CPUs, with smaller process sizes (e.g., 7nm, 10nm) becoming more common in recent years.
- **Transistor counts** have a similar skew to CPUs, with a few high-end GPUs boasting over 10,000 million transistors.
- **Die size** shows a right skew, with most GPUs between 200–400 mm^2, but some reaching as high as 1000 mm^2.
- **Performance Metrics**:
  - **Boost clock speeds** and **base clock speeds** for GPUs generally fall between 1000–2000 MHz, with a long tail extending to higher values, representing premium or overclocked models.
  - **Memory size** shows a diverse range, with a peak at 4 GB and 8 GB, but with modern GPUs increasingly offering higher memory configurations (up to 24 GB).
  - **TDP** values for GPUs show a right-skewed distribution, with most GPUs consuming less than 300 watts, but extreme cases reaching as high as 800 watts, especially for top-tier models.

---

2. Price Trends Over Time (Original vs Inflation-Adjusted Prices)

CPU Data

- The pricing plots show the average original launch prices and inflation-adjusted prices for CPUs from 2004 to 2024.

- **Price Stability**: Between 2004 and 2020, there is relative stability in both the original and inflation-adjusted prices, with the inflation-adjusted prices staying higher than the original prices due to market and inflationary factors.
- **Price Surge in 2021–2023**: A sharp increase in both original and inflation-adjusted prices is observed starting in 2021, peaking in 2023. This likely correlates with major advancements in processor technology and global supply chain disruptions, such as the semiconductor shortage.
- **Price Drop in 2024**: Interestingly, there is a significant drop in 2024, which may either be a lack of full 2024 release data and/or a market correction after the inflated prices of the previous years.

GPU Data

- The GPU data presents a similar pattern of inflation-adjusted prices consistently being higher than original prices.
- **Volatility in GPU Pricing**: Unlike CPUs, GPU prices show higher volatility over the years, with notable price spikes in 2008, 2016, and 2020. The post-2020 period also sees a notable peak, similar to CPUs.
- **High-end GPUs**: Top-tier GPUs (priced above $1600) show pronounced price spikes, particularly in years like 2011 and 2018, suggesting that premium GPUs are disproportionately impacted by market fluctuat
899 in 2024 dollars, contrasts with the Nvidia RTX 4090, which remains priced at $1599 in 2024.

---

3. Price Range Analysis for CPUs and GPUs

- **CPU and GPU Price Bands**:
  - Both CPUs and GPUs were divided into price bands (0–250, 250–500, 500–1000, $1000+) and averaged within those ranges. Within each range, we analyzed the original and inflation-adjusted prices over time.
  - **High-End Segment**: The $1000 + segment for CPUs shows a pronounced spike in 2021 – 2023, which was ex . For GPUs, the
1600+ segment also experiences high volatility, particularly for top-tier gaming and professional GPUs.

---

4. Correlation Analysis

CPU Correlation Matrix

- **High Correlations**: Notable positive correlations are observed between:

- Transistor count and die size (0.70) — as more transistors are packed into a processor, the die size tends to increase, reflecting advancements in chip design.
- Core and thread counts (0.92) — higher core counts are naturally associated with more threads.
- Die size and TDP (0.70) — larger die sizes correlate with higher power consumption, which is intuitive as more transistors and cores demand greater power.

Negative Correlations:

- Process size and transistor count (-0.39) — as the process size decreases (smaller nanometers), the transistor count increases, reflecting advances in semiconductor technology (Moore's Law).

GPU Correlation Matrix

- **Strong Positive Correlations**:
  - Shading units, TMUs, ROPs, and SM count (ranging from 0.85 to 0.93) — these features are strongly linked as they all contribute to the rendering capabilities of the GPU.
  - Memory bus, bandwidth, and memory size — larger memory buses and higher bandwidths tend to come with larger memory sizes, which is a clear indicator of the focus on improving GPU performance.
- **Performance Features**:
  - Boost clock speeds are strongly correlated with base clock speeds (0.99), as these are often close in magnitude and work together to define a GPU's overall clock performance.

Inverse Correlations:

- Process size and various performance metrics (like boost clock and shading units) show negative correlations, indicating that newer manufacturing processes (smaller nanometer sizes) allow for more performance improvements while reducing the physical size of transistors.

Key Observations and Insights

1. **Technology Trends**:
   - Smaller process sizes correlate with performance gains (higher clock speeds, transistor counts, etc.), as expected from Moore's Law.
   - Both CPUs and GPUs show advancements in core and thread counts, shading units, and clock speeds, driven by the need for higher performance in computing and gaming.

2. **Price Trends**:
    - Prices remained relatively stable until 2021, after which a significant surge occurred, particularly in the high-end segment, driven by new technologies, market demand, and supply chain challenges.
    - Inflation-adjusted prices suggest that while nominal prices have increased, the real value of these components has been more consistent over time, especially for mid-range models.
3. **Feature Importance for Price Prediction**:
    - Features such as transistor count, die size, clock speed, and core count are highly correlated with price and will likely play an important role in predictive modeling. These attributes will be key inputs in predicting the launch price of CPUs and GPUs.

EDA Conclusion

The EDA provided valuable insights into the historical trends of CPU and GPU attributes, their pricing behavior, and the relationships between technical specifications. The correlation analysis revealed important relationships that will inform the feature selection for predicting CPU and GPU prices. In subsequent steps, we will leverage these insights to build predictive models that can accurately forecast future component prices based on their technical specifications.

Models

Model Selection and Justification

In this section, we will build and evaluate different machine learning models to predict CPU and GPU prices. The chosen models include a mix of linear and non-linear approaches, ensuring that we capture both simple and complex relationships in the data.

Given the nature of the problem—predicting prices from a variety of features like clock speed, core count, transistor count, and others—our dataset likely has both linear and non-linear patterns. Therefore, we need a combination of baseline models and more advanced models to compare performance.

To strike a balance between interpretability, computational complexity, and predictive power, we will use:

1. **Linear Regression**: A basic linear model to serve as a benchmark and provide easy-to-interpret insights.
2. **Random Forest**: A robust tree-based model that handles non-linear interactions and provides feature importance scores.
3. **XGBoost**: A more powerful and efficient boosting model designed to handle large datasets and complex relationships, which should improve prediction accuracy when tuned.
4. **Gradient Boosting**: Another boosting algorithm that iteratively improves performance by focusing on correcting the errors of previous models, making it particularly effective for capturing complex patterns in smaller datasets.

Below is a review of the models covered in the course, along with an additional model I plan to consider for this section, highlighting their respective pros and cons:

Model	Pros	Cons	Link/Source
Simple Linear Regression	- Easy to interpret - Works well when data has a linear relationship	- Fails with non-linear data - Sensitive to outliers	Linear Regression - Wikipedia Advantages and Disadvantages - GeeksforGeeks
Multiple Linear Regression	- Can model more complex relationships with multiple features - Relatively easy to interpret	- Multicollinearity can cause issues - Assumes linearity and homoscedasticity	A Guide to Multiple Regression Using Statsmodels Advantages and Disadvantages - GeeksforGeeks
Polynomial Regression	- Can model non-linear relationships - Simple to implement	- Prone to overfitting - Sensitive to outliers	Polynomial Regression - Scikit-Learn Advantages and Disadvantages - GeeksforGeeks
Logistic Regression	- Works well for binary classification - Easy to implement and interpret	- Assumes a linear relationship between features - Not good for complex decision boundaries	Logistic Regression - Scikit-Learn Advantages and Disadvantages - GeeksforGeeks
K-Nearest Neighbors (KNN)	- Simple and intuitive - Good for small datasets and non-linear relationships	- Computationally expensive on large datasets - Sensitive to the choice of k	KNN Algorithm - IBM KNN Algorithm - Towards Data Science
Decision Trees	- Easy to visualize and interpret - Handles non-linear relationships well	- Prone to overfitting without pruning - Sensitive to small changes in data	Decision Trees - Towards Data Science Advantages and Disadvantages - GeeksforGeeks

Model	Pros	Cons	Link/Source
Random Forest	- Reduces overfitting by averaging multiple trees - Good performance on large datasets	- Harder to interpret than individual trees - Can be slow on very large datasets	Random Forest - Scikit-Learn Advantages and Disadvantages - GeeksforGeeks
Bagging (Ensemble)	- Reduces variance and overfitting - Good for improving unstable models (like decision trees)	- Computationally intensive - Not effective for small datasets	Bagging - IBM Bagging - Machine Learning Mastery
Boosting (Ensemble)	- Good at handling bias - Great for improving weak models like decision trees	- Prone to overfitting on noisy data - Requires careful tuning of hyperparameters	Boosting - IBM Boosting - GeeksforGeeks
AdaBoost	- Reduces bias significantly - Effective for binary classification problems	- Sensitive to noisy data - Requires careful tuning	AdaBoost - Paperspace AdaBoost - Scikit-Learn
Gradient Boosting	- Excellent accuracy - Can handle both regression and classification tasks	- Prone to overfitting without tuning - Requires longer training times	Gradient Boosting - Scikit-Learn Gradient Boosting - Digital Ocean
Support Vector Machines (SVM)	- Works well with high-dimensional data - Effective when there is a clear margin of separation	- Requires careful tuning of the kernel - Computationally intensive on large datasets	SVM - Scikit-Learn Advantages and Disadvantages - GeeksforGeeks
XGBoost	- Extremely efficient - Great at handling large datasets - Can handle missing values	- Can be prone to overfitting without tuning - Longer training times for small datasets	XGBoost - Towards Data Science XGBoost - Krayonnz XGBoost - NVIDIA XGBoost - Kaggle

Also this article summarizes a good number of the different models: https://jespublication.com/upload/2021-V12I1035.pdf.

Model Rationale

Why Linear Regression?

We start with Linear Regression because it's a simple, interpretable baseline. It's helpful for understanding feature importance and as a comparative benchmark. However, due to the complexity of relationships in this dataset, we anticipate it may not perform well on its own.

Why Random Forest?
Next, we use Random Forest, which is excellent at capturing non-linear relationships and interactions between features. It also provides insights into feature importance and avoids the issues of multicollinearity by randomly selecting subsets of features for each tree.

Why XGBoost?
Also, we apply XGBoost, a state-of-the-art boosting algorithm, known for its efficiency and accuracy in large datasets. XGBoost applies regularization, helping to avoid overfitting, and is well-suited to complex tasks like this.

Why Gradient Boosting? Additionally, Gradient Boosting is introduced, which iteratively builds models that correct the errors of previous models. It is powerful for capturing complex patterns in smaller datasets and is less prone to overfitting with proper tuning. This makes it well-suited for our GPU price prediction task, where the relationships between features are intricate.

These four models allow us to test a variety of approaches, from simple to advanced, and ensure that both linear and non-linear patterns are well captured in our predictions. By comparing them, we can assess which model best captures the pricing trends in the CPU and GPU markets.

Step 1: Data Preparation for CPU and GPU

Use inflation adjusted 2024 prices for the data prediction.

Addressing multicollinearity and Feature Engineering

See https://www.geeksforgeeks.org/detecting-multicollinearity-with-vif-python/ and https://online.stat.psu.edu/stat462/node/180/ for more details on the VIF tests.

When we run the code below, we will see the the following data has high VIF values indicating that they are highly collinear. VIF of greater than 5 is an indication of multicollinearity. We will use feature engineering to combine highly correlated data to reduce the VIF values.

In [10]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# NaN Imputation

# Load the CPU and GPU datasets
cpu_data = pd.read_excel('../src/cpu_data_cleaned.xlsx')
gpu_data = pd.read_excel('../src/gpu_data_cleaned.xlsx')

# Replace 'Not Provided' with NaN in both datasets
cpu_data.replace('Not Provided', pd.NA, inplace=True)
```

```
gpu_data.replace('Not Provided', pd.NA, inplace=True)

# Define the numerical columns for CPU and GPU before feature engineering
cpu_numerical_columns = [
    'Physical - Process Size (in nm)', 'Physical - Transistors (in millions)', 'Ph
    'Performance - Frequency (in GHz)', 'Performance - Turbo Clock (up to in GHz)'
    'Core Config - # of Cores', 'Core Config - # of Threads', 'Cache L1 Size (in K
    'Cache L2 Size (in MB)', 'Cache L3 Size (in MB)'
]

gpu_numerical_columns = [
    "Process Size (in nm)", "Transistors (in millions)", "Density (in M / mm²)", '
    "Base Clock (in MHz)", "Boost Clock (in MHz)", "Memory Clock (in MHz)", "Effec
    "Memory Size (in GB)", "Memory Bus (in bit)", "Bandwidth (in GB/s)", "Shading
    "SM Count", "Tensor Cores", "RT Cores", "L1 Cache (in KB)", "L2 Cache (in MB)'
    "Texture Rate (in Gtexel/s)", "FP16 (half, in TFLOPS)", "FP64 (double, in TFLO
    "TDP (in Watts)"
]

# Convert numerical columns to numeric (invalid parsing results in NaN)
cpu_data[cpu_numerical_columns] = cpu_data[cpu_numerical_columns].apply(pd.to_nume
gpu_data[gpu_numerical_columns] = gpu_data[gpu_numerical_columns].apply(pd.to_nume

# Add "Release Year" feature
cpu_data['Release Year'] = pd.to_datetime(cpu_data['Processor - Release Date'], er
gpu_data['Release Year'] = pd.to_datetime(gpu_data['Release Date'], errors='coerce

# Fill NaN values with column means
cpu_data[cpu_numerical_columns] = cpu_data[cpu_numerical_columns].fillna(cpu_data[
gpu_data[gpu_numerical_columns] = gpu_data[gpu_numerical_columns].fillna(gpu_data[

# Replace 'Not Provided' in the target columns and convert to numeric
cpu_data['Processor - Launch Price Inflation Adjusted for 2024 (in USD)'] = pd.to_
    cpu_data['Processor - Launch Price Inflation Adjusted for 2024 (in USD)'].repl
)
gpu_data['GPU Launch Price adjusted for inflation in 2024 (in USD)'] = pd.to_numer
    gpu_data['GPU Launch Price adjusted for inflation in 2024 (in USD)'].replace(
)

# Drop rows with NaN target values
cpu_data.dropna(subset=['Processor - Launch Price Inflation Adjusted for 2024 (in
gpu_data.dropna(subset=['GPU Launch Price adjusted for inflation in 2024 (in USD)'

# Separate features (X) and target variable (y)
X_cpu, y_cpu = cpu_data[cpu_numerical_columns], cpu_data['Processor - Launch Price
X_gpu, y_gpu = gpu_data[gpu_numerical_columns], gpu_data['GPU Launch Price adjuste
```

In [11]:

```
import pandas as pd
from statsmodels.stats.outliers_influence import variance_inflation_factor

def calculate_vif(X):
    vif_data = pd.DataFrame()
    vif_data["feature"] = X.columns
    vif_data["VIF"] = [variance_inflation_factor(X.values, i) for i in range(X.sha
    return vif_data
```

```
# Calculate VIF for CPU features
vif_cpu = calculate_vif(X_cpu)

# Calculate VIF for GPU features
vif_gpu = calculate_vif(X_gpu)

print(vif_cpu)
print(vif_gpu)
```

```
                                  feature        VIF
0           Physical - Process Size (in nm)   2.161303
1       Physical - Transistors (in millions)   4.819218
2              Physical - Die Size(in  mm²)   4.733996
3          Performance - Frequency (in GHz)   4.976802
4   Performance - Turbo Clock (up to in GHz)   1.048029
5              Performance - TDP (in Watts)  13.143453
6                  Core Config - # of Cores  24.321639
7                Core Config - # of Threads  36.412497
8                     Cache L1 Size (in KB)   2.557058
9                     Cache L2 Size (in MB)   1.012595
10                    Cache L3 Size (in MB)   3.987724
                      feature         VIF
0         Process Size (in nm)    8.980950
1    Transistors (in millions)   41.203753
2         Density (in M / mm²)   28.176967
3            Die Size (in mm²)   27.512796
4          Base Clock (in MHz)   90.764824
5         Boost Clock (in MHz)  145.694340
6        Memory Clock (in MHz)   31.277742
7    Effective Speed (in Gbps)   42.182631
8          Memory Size (in GB)    5.244069
9          Memory Bus (in bit)    8.217193
10         Bandwidth (in GB/s)   40.165380
11               Shading Units   44.360782
12                        TMUs   86.699226
13                        ROPs   95.827209
14                    SM Count   20.450912
15                Tensor Cores   61.136763
16                    RT Cores   59.891921
17           L1 Cache (in KB)   17.091801
18           L2 Cache (in MB)    7.752876
19    Pixel Rate (in GPixel/s)  166.607178
20  Texture Rate (in Gtexel/s)  158.380949
21      FP16 (half, in TFLOPS)   13.879779
22    FP64 (double, in TFLOPS)    4.369113
23              TDP (in Watts)    7.970430
```

We will perform feature engineering to address the collinearity.

In [12]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# NaN Imputation

# Load the CPU and GPU datasets
```

```
cpu_data = pd.read_excel('../src/cpu_data_cleaned.xlsx')
gpu_data = pd.read_excel('../src/gpu_data_cleaned.xlsx')

# Replace 'Not Provided' with NaN in both datasets
cpu_data.replace('Not Provided', pd.NA, inplace=True)
gpu_data.replace('Not Provided', pd.NA, inplace=True)

# Define the numerical columns for CPU and GPU before feature engineering
cpu_numerical_columns = [
    'Physical - Process Size (in nm)', 'Physical - Transistors (in millions)', 'Ph
    'Performance - Frequency (in GHz)', 'Performance - Turbo Clock (up to in GHz)'
    'Core Config - # of Cores', 'Core Config - # of Threads', 'Cache L1 Size (in K
    'Cache L2 Size (in MB)', 'Cache L3 Size (in MB)'
]

gpu_numerical_columns = [
    "Process Size (in nm)", "Transistors (in millions)", "Density (in M / mm²)", '
    "Base Clock (in MHz)", "Boost Clock (in MHz)", "Memory Clock (in MHz)", "Effec
    "Memory Size (in GB)", "Memory Bus (in bit)", "Bandwidth (in GB/s)", "Shading
    "SM Count", "Tensor Cores", "RT Cores", "L1 Cache (in KB)", "L2 Cache (in MB)'
    "Texture Rate (in Gtexel/s)", "FP16 (half, in TFLOPS)", "FP64 (double, in TFLO
    "TDP (in Watts)"
]

# Convert numerical columns to numeric (invalid parsing results in NaN)
cpu_data[cpu_numerical_columns] = cpu_data[cpu_numerical_columns].apply(pd.to_nume
gpu_data[gpu_numerical_columns] = gpu_data[gpu_numerical_columns].apply(pd.to_nume

# Add "Release Year" feature
cpu_data['Release Year'] = pd.to_datetime(cpu_data['Processor - Release Date'], er
gpu_data['Release Year'] = pd.to_datetime(gpu_data['Release Date'], errors='coerce

# Feature Engineering: Create interaction terms
cpu_data['Cores_Threads_Product'] = cpu_data['Core Config - # of Cores'] * cpu_dat
cpu_data['Frequency_TDP_Ratio'] = cpu_data['Performance - Frequency (in GHz)'] / c
gpu_data['Base_Boost_Clock_Product'] = gpu_data['Base Clock (in MHz)'] * gpu_data[
gpu_data['Bandwidth_Memory_Ratio'] = gpu_data['Bandwidth (in GB/s)'] / gpu_data['M
gpu_data['Graphics_Processing_Power'] = gpu_data['Shading Units'] + gpu_data['TMUs
gpu_data['Transistor_Density'] = (gpu_data['Transistors (in millions)'] * gpu_data
gpu_data['Multiplied_Speed_Rate'] = (gpu_data['Effective Speed (in Gbps)'] * gpu_c

# Updated lists of numerical columns after feature engineering
cpu_numerical_columns = [
    'Physical - Process Size (in nm)', 'Physical - Transistors (in millions)', 'Ph
    'Performance - Turbo Clock (up to in GHz)', 'Cache L1 Size (in KB)', 'Cache L2
    'Cache L3 Size (in MB)', 'Cores_Threads_Product', 'Frequency_TDP_Ratio', 'Rele
]

gpu_numerical_columns = [
    "Process Size (in nm)", "Memory Clock (in MHz)", "Memory Bus (in bit)",
    "SM Count", "Tensor Cores", "RT Cores", "L1 Cache (in KB)",
    "L2 Cache (in MB)", "FP16 (half, in TFLOPS)", #"FP32 (float, in TFLOPS)",
    "FP64 (double, in TFLOPS)", "TDP (in Watts)", 'Base_Boost_Clock_Product',
    'Bandwidth_Memory_Ratio', 'Graphics_Processing_Power', 'Transistor_Density',
    'Release Year', 'Multiplied_Speed_Rate'  # New feature created by multiplying
]
```

```
# Drop highly collinear features
cpu_data.drop(['Core Config - # of Cores', 'Core Config - # of Threads', 'Performa
gpu_data.drop(['Base Clock (in MHz)', 'Boost Clock (in MHz)', 'Bandwidth (in GB/s)
               'Transistors (in millions)', 'Density (in M / mm²)', 'Die Size (in
               'Shading Units', 'TMUs', 'ROPs'], axis=1, inplace=True)
gpu_data.drop(['Effective Speed (in Gbps)', 'Pixel Rate (in GPixel/s)', 'Texture F

# Fill NaN values with column means
cpu_data[cpu_numerical_columns] = cpu_data[cpu_numerical_columns].fillna(cpu_data[
gpu_data[gpu_numerical_columns] = gpu_data[gpu_numerical_columns].fillna(gpu_data[

# Replace 'Not Provided' in the target columns and convert to numeric
cpu_data['Processor - Launch Price Inflation Adjusted for 2024 (in USD)'] = pd.to_
    cpu_data['Processor - Launch Price Inflation Adjusted for 2024 (in USD)'].repl
)
gpu_data['GPU Launch Price adjusted for inflation in 2024 (in USD)'] = pd.to_numer
    gpu_data['GPU Launch Price adjusted for inflation in 2024 (in USD)'].replace(
)

# Drop rows with NaN target values
cpu_data.dropna(subset=['Processor - Launch Price Inflation Adjusted for 2024 (in
gpu_data.dropna(subset=['GPU Launch Price adjusted for inflation in 2024 (in USD)'

# Separate features (X) and target variable (y)
X_cpu, y_cpu = cpu_data[cpu_numerical_columns], cpu_data['Processor - Launch Price
X_gpu, y_gpu = gpu_data[gpu_numerical_columns], gpu_data['GPU Launch Price adjuste

# Split CPU and GPU data into training and test sets
X_cpu_train, X_cpu_test, y_cpu_train, y_cpu_test = train_test_split(X_cpu, y_cpu,
X_gpu_train, X_gpu_test, y_gpu_train, y_gpu_test = train_test_split(X_gpu, y_gpu,

# Separate scalers for CPU and GPU
scaler_cpu = StandardScaler()
scaler_gpu = StandardScaler()

X_cpu_train_scaled = scaler_cpu.fit_transform(X_cpu_train)
X_cpu_test_scaled = scaler_cpu.transform(X_cpu_test)
X_gpu_train_scaled = scaler_gpu.fit_transform(X_gpu_train)
X_gpu_test_scaled = scaler_gpu.transform(X_gpu_test)

# Sanity check
print(cpu_data[cpu_numerical_columns].isnull().sum())
print(gpu_data[gpu_numerical_columns].isnull().sum())
print(X_cpu_train_scaled[:5])  # First 5 rows of scaled CPU training data
print(X_gpu_train_scaled[:5])  # First 5 rows of scaled GPU training data
print(f"CPU Train shape: {X_cpu_train.shape}, CPU Test shape: {X_cpu_test.shape}")
print(f"GPU Train shape: {X_gpu_train.shape}, GPU Test shape: {X_gpu_test.shape}")
```

```
Physical - Process Size (in nm)             0
Physical - Transistors (in millions)        0
Physical - Die Size(in  mm²)                0
Performance - Turbo Clock (up to in GHz)    0
Cache L1 Size (in KB)                       0
Cache L2 Size (in MB)                       0
Cache L3 Size (in MB)                       0
Cores_Threads_Product                       0
Frequency_TDP_Ratio                         0
Release Year                                0
dtype: int64
Process Size (in nm)         0
Memory Clock (in MHz)        0
Memory Bus (in bit)          0
SM Count                     0
Tensor Cores                 0
RT Cores                     0
L1 Cache (in KB)             0
L2 Cache (in MB)             0
FP16 (half, in TFLOPS)       0
FP64 (double, in TFLOPS)     0
TDP (in Watts)               0
Base_Boost_Clock_Product     0
Bandwidth_Memory_Ratio       0
Graphics_Processing_Power    0
Transistor_Density           0
Release Year                 0
Multiplied_Speed_Rate        0
dtype: int64
[[ 2.61369601 -0.47166231 -0.24864963  0.221408    0.49419905 -0.04567463
  -0.13508384 -0.2918121  -0.50076742 -1.80716697]
 [-0.57327052 -0.15237054 -0.30044158 -0.11859785 -0.10818095 -0.03072162
  -0.22935517 -0.22898335 -0.40576528  0.68305869]
 [-0.41392219 -0.15237054 -0.24106     0.221408   -0.30897428 -0.09053364
  -0.42331533 -0.29020111  0.8416273   0.68305869]
 [-0.57327052 -0.15237054 -0.43510066 -0.12740493 -0.10818095 -0.03072162
  -0.25360019 -0.26442521  0.25690268  0.84907373]
 [ 2.61369601 -0.47166231 -0.24864963  0.221408    0.49419905 -0.04567463
  -0.13508384 -0.2918121  -0.60133359 -1.80716697]]
[[-0.59580543  1.2485616  -0.37756262 -1.39199348  0.09152276 -0.12932067
  -0.15763145 -0.32533952 -1.34358478 -0.70189772 -1.36771903 -0.35757906
  -1.03549968 -0.76436326 -0.03471069  0.33777447 -0.29767566]
 [-0.88753474 -0.44264933 -0.47142424 -1.2653552  -4.22415169 -0.12932067
   1.73219829 -0.35293118 -1.34734662 -0.35895385 -1.41205997 -0.35757906
  -1.01455633 -0.65739298 -0.03471069  1.694468   -0.29876314]
 [-0.99693323  1.53219397 -0.37756262 -0.12561069 -3.1413894  -2.0079349
   1.73219829  0.94387666 -0.55662113 -0.51094033 -0.87996869  1.40951508
  -0.86861271  0.38470456  4.2210027   1.88828136 -0.05845546]
 [-0.01234681 -0.26817329  0.0183415  -0.07775239  0.09152276 -0.12932067
  -0.02785076  0.15283548 -0.02841241  0.4068315  -0.92430963 -0.35757906
  -0.29461913 -0.76781392 -0.37509792 -0.43747898 -0.09924876]
 [-0.15821147  0.20928179 -0.18983938 -0.07775239  0.09152276 -0.12932067
  -0.91356334 -0.32533952 -0.02841241  0.19855918  0.49460046 -0.35757906
  -0.32120194 -0.05352852 -0.33513763 -0.43747898 -0.28640196]]
CPU Train shape: (935, 10), CPU Test shape: (234, 10)
GPU Train shape: (381, 17), GPU Test shape: (96, 17)
```

Let's rerun the VIF tests again!

In [13]:
```
from statsmodels.stats.outliers_influence import variance_inflation_factor

# Calculate VIF for CPU data
vif_cpu = calculate_vif(pd.DataFrame(X_cpu_train_scaled, columns=X_cpu.columns))
print("CPU VIF after feature engineering:")
print(vif_cpu)

# Calculate VIF for GPU data
vif_gpu = calculate_vif(pd.DataFrame(X_gpu_train_scaled, columns=X_gpu.columns))
print("GPU VIF after feature engineering:")
print(vif_gpu)
```

```
CPU VIF after feature engineering:
                                  feature       VIF
0           Physical - Process Size (in nm)  4.644761
1      Physical - Transistors (in millions)  3.607686
2              Physical - Die Size(in  mm²)  1.631641
3  Performance - Turbo Clock (up to in GHz)  1.011137
4                     Cache L1 Size (in KB)  1.057586
5                     Cache L2 Size (in MB)  1.002191
6                     Cache L3 Size (in MB)  3.511862
7                     Cores_Threads_Product  1.876104
8                       Frequency_TDP_Ratio  1.331627
9                              Release Year  4.794624
GPU VIF after feature engineering:
                      feature        VIF
0        Process Size (in nm)   7.505184
1       Memory Clock (in MHz)   3.489595
2         Memory Bus (in bit)   2.786291
3                    SM Count   5.221644
4                Tensor Cores   2.570342
5                    RT Cores   5.505437
6            L1 Cache (in KB)   3.748301
7            L2 Cache (in MB)   6.997593
8      FP16 (half, in TFLOPS)   4.772599
9    FP64 (double, in TFLOPS)   2.157520
10             TDP (in Watts)   1.728315
11   Base_Boost_Clock_Product   3.517461
12     Bandwidth_Memory_Ratio   1.388175
13  Graphics_Processing_Power   8.703025
14         Transistor_Density   5.903773
15               Release Year  14.155353
16      Multiplied_Speed_Rate  10.811172
```

Step 2: Build and Train the CPU and GPU price predictor model

Now that we have a useable and clean data, let us train and iterate our supervised machine learning models!

Linear Regression

In [14]:
```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, root_mean_squared_error, r2_score

# Initialize Linear Regression model for CPU
linear_model_cpu = LinearRegression()
linear_model_cpu.fit(X_cpu_train_scaled, y_cpu_train)

# Make predictions for CPU
y_pred_cpu_lr = linear_model_cpu.predict(X_cpu_test_scaled)



# Initialize Linear Regression model for GPU
linear_model_gpu = LinearRegression()
linear_model_gpu.fit(X_gpu_train_scaled, y_gpu_train)

# Make predictions for GPU
y_pred_gpu_lr = linear_model_gpu.predict(X_gpu_test_scaled)
```

Random Forest

In [15]:
```
from sklearn.ensemble import RandomForestRegressor

# Initialize Random Forest model for CPU
rf_model_cpu = RandomForestRegressor(n_estimators=100, random_state=42)
rf_model_cpu.fit(X_cpu_train, y_cpu_train)

# Make predictions for CPU
y_pred_cpu_rf = rf_model_cpu.predict(X_cpu_test)



# Initialize Random Forest model for GPU
rf_model_gpu = RandomForestRegressor(n_estimators=100, random_state=42)
rf_model_gpu.fit(X_gpu_train, y_gpu_train)

# Make predictions for GPU
y_pred_gpu_rf = rf_model_gpu.predict(X_gpu_test)
```

XGBoost

In [16]:
```
import xgboost as xgb

# Initialize XGBoost model for CPU
xgb_model_cpu = xgb.XGBRegressor(n_estimators=100, learning_rate=0.1, max_depth=5,
xgb_model_cpu.fit(X_cpu_train, y_cpu_train)

# Make predictions for CPU
y_pred_cpu_xgb = xgb_model_cpu.predict(X_cpu_test)



# Initialize XGBoost model for GPU
xgb_model_gpu = xgb.XGBRegressor(n_estimators=100, learning_rate=0.1, max_depth=5,
```

```
xgb_model_gpu.fit(X_gpu_train, y_gpu_train)

# Make predictions for GPU
y_pred_gpu_xgb = xgb_model_gpu.predict(X_gpu_test)
```

Gradient Boosting (GPU only)

In [17]:
```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

# Initialize Gradient Boosting Regressor
gbr_gpu = GradientBoostingRegressor(n_estimators=100, learning_rate=0.1, max_depth

# Train the model on GPU data
gbr_gpu.fit(X_gpu_train_scaled, y_gpu_train)

# Predict on the test data
y_gpu_pred_gbr = gbr_gpu.predict(X_gpu_test_scaled)

# Evaluate the Gradient Boosting model
gpu_mae_gbr = mean_absolute_error(y_gpu_test, y_gpu_pred_gbr)
gpu_rmse_gbr = root_mean_squared_error(y_gpu_test, y_gpu_pred_gbr)  # Root mean sq
gpu_r2_gbr = r2_score(y_gpu_test, y_gpu_pred_gbr)
```

Printing the benchmark for the default models:

In [18]:
```
# Organize CPU and GPU results by model for easier readability

# Evaluate CPU Linear Regression
print("\nCPU Linear Regression Results:")
print(f"MAE: {mean_absolute_error(y_cpu_test, y_pred_cpu_lr):.4f} | RMSE: {root_me

# Evaluate GPU Linear Regression
print("\nGPU Linear Regression Results:")
print(f"MAE: {mean_absolute_error(y_gpu_test, y_pred_gpu_lr):.4f} | RMSE: {root_me

# Evaluate CPU Random Forest
print("\nCPU Random Forest Results:")
print(f"MAE: {mean_absolute_error(y_cpu_test, y_pred_cpu_rf):.4f} | RMSE: {root_me

# Evaluate GPU Random Forest
print("\nGPU Random Forest Results:")
print(f"MAE: {mean_absolute_error(y_gpu_test, y_pred_gpu_rf):.4f} | RMSE: {root_me

# Evaluate CPU XGBoost
print("\nCPU XGBoost Results:")
print(f"MAE: {mean_absolute_error(y_cpu_test, y_pred_cpu_xgb):.4f} | RMSE: {root_r

# Evaluate GPU XGBoost
print("\nGPU XGBoost Results:")
print(f"MAE: {mean_absolute_error(y_gpu_test, y_pred_gpu_xgb):.4f} | RMSE: {root_r

# Evaluate GPU Gradient Boosting
```

```
print("\nGPU Gradient Boosting Results:")
print(f"MAE: {gpu_mae_gbr:.4f} | RMSE: {gpu_rmse_gbr:.4f} | R²: {gpu_r2_gbr:.4f}")
```

```
CPU Linear Regression Results:
MAE: 560.0673 | RMSE: 1171.8899 | R²: 0.7196

GPU Linear Regression Results:
MAE: 1326.8062 | RMSE: 1804.8386 | R²: 0.5061

CPU Random Forest Results:
MAE: 343.9683 | RMSE: 686.5323 | R²: 0.9038

GPU Random Forest Results:
MAE: 813.7038 | RMSE: 1574.1856 | R²: 0.6242

CPU XGBoost Results:
MAE: 341.6049 | RMSE: 642.1243 | R²: 0.9158

GPU XGBoost Results:
MAE: 867.7665 | RMSE: 1771.3949 | R²: 0.5242

GPU Gradient Boosting Results:
MAE: 679.3741 | RMSE: 1201.7698 | R²: 0.7810
```

Step 3: Refinement and tuning of the models

Linear Regression Tuning

- **Parameter Tuning**: Uses `GridSearchCV` to fine-tune the `alpha` parameter (regularization strength) for Ridge regression.
- **Cross-Validation**: Performs 5-fold cross-validation to find the optimal `alpha` for minimizing the mean absolute error (MAE).
- **Separate Models**: Fine-tunes separate Ridge regression models for CPU and GPU datasets.
- **Performance Metrics**: After finding the best model, it evaluates performance on test data using MAE, RMSE, and R^2 metrics.
- **Output**: Prints the performance results for both CPU and GPU datasets.

In [19]:
```
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_absolute_error, root_mean_squared_error, r2_score

# Set up the parameter grid for alpha (regularization strength)
ridge_params = {'alpha': [0.01, 0.1, 1.0, 10.0, 100.0]}

# Initialize Ridge regression for both CPU and GPU
ridge_cpu = Ridge()
ridge_gpu = Ridge()

# Grid search for best alpha (CPU)
ridge_grid_cpu = GridSearchCV(ridge_cpu, ridge_params, cv=5, scoring='neg_mean_abs
ridge_grid_cpu.fit(X_cpu_train_scaled, y_cpu_train)
```

```
# Grid search for best alpha (GPU)
ridge_grid_gpu = GridSearchCV(ridge_gpu, ridge_params, cv=5, scoring='neg_mean_abs
ridge_grid_gpu.fit(X_gpu_train_scaled, y_gpu_train)

# Get best estimators
best_ridge_cpu = ridge_grid_cpu.best_estimator_
best_ridge_gpu = ridge_grid_gpu.best_estimator_

# Predict on CPU test data
y_cpu_pred_ridge = best_ridge_cpu.predict(X_cpu_test_scaled)

# Predict on GPU test data
y_gpu_pred_ridge = best_ridge_gpu.predict(X_gpu_test_scaled)

# Calculate performance metrics for CPU
cpu_mae_ridge = mean_absolute_error(y_cpu_test, y_cpu_pred_ridge)
cpu_rmse_ridge = root_mean_squared_error(y_cpu_test, y_cpu_pred_ridge)
cpu_r2_ridge = r2_score(y_cpu_test, y_cpu_pred_ridge)

# Calculate performance metrics for GPU
gpu_mae_ridge = mean_absolute_error(y_gpu_test, y_gpu_pred_ridge)
gpu_rmse_ridge = root_mean_squared_error(y_gpu_test, y_gpu_pred_ridge)
gpu_r2_ridge = r2_score(y_gpu_test, y_gpu_pred_ridge)

# Print the results for CPU
print(f"Ridge Regression (CPU) - MAE: {cpu_mae_ridge:.4f} | RMSE: {cpu_rmse_ridge

# Print the results for GPU
print(f"Ridge Regression (GPU) - MAE: {gpu_mae_ridge:.4f} | RMSE: {gpu_rmse_ridge
```

```
Ridge Regression (CPU) - MAE: 559.4776 | RMSE: 1168.7277 | R²: 0.7211
Ridge Regression (GPU) - MAE: 1300.9003 | RMSE: 1788.5104 | R²: 0.5150
```

Random Forest Tuning

- **Parameter Tuning**:
  - **RandomizedSearchCV**: Uses a random search strategy to sample 50 different combinations of hyperparameters for the Random Forest model, based on the provided `param_dist` grid.
  - **GridSearchCV**: For GPU data, it exhaustively searches over a grid of hyperparameters specified in `param_grid_gpu`, including a log-scaled range for `n_estimators` and `max_depth`.
- **Cross-Validation**: Both `RandomizedSearchCV` and `GridSearchCV` use 3-fold cross-validation to evaluate model performance for each set of hyperparameters.
- **Performance Evaluation**: After finding the best Random Forest model for both CPU and GPU data, it evaluates the model's performance using metrics such as MAE, RMSE, and R^2.
- **Parallel Processing**: Both searches (`RandomizedSearchCV` and `GridSearchCV`) utilize all available CPU cores (`n_jobs=-1`) to speed up the search process.

- **Output**: Prints the best model's performance for CPU and GPU data, along with the optimal hyperparameters for the GPU model.

In [20]:
```
from sklearn.model_selection import RandomizedSearchCV, GridSearchCV
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error, root_mean_squared_error, r2_score
import numpy as np


# Define the parameter grid for RandomizedSearchCV
param_dist = {
    'n_estimators': [100, 200, 300, 500],
    'max_depth': [10, 20, 30, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'bootstrap': [True, False]
}

# Initialize the Random Forest Regressor
rf = RandomForestRegressor(random_state=42)

# Set up RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=rf,
    param_distributions=param_dist,
    n_iter=50,  # Number of parameter settings sampled
    cv=3,  # 3-fold cross validation
    verbose=2,
    random_state=42,
    n_jobs=-1  # Use all available cores
)

# Perform random search on CPU data
random_search.fit(X_cpu_train_scaled, y_cpu_train)

# Get the best estimator and predict on test data
best_rf_cpu = random_search.best_estimator_
y_pred_cpu_rf = best_rf_cpu.predict(X_cpu_test_scaled)

# Evaluate the tuned Random Forest model
cpu_mae_rf = mean_absolute_error(y_cpu_test, y_pred_cpu_rf)
cpu_rmse_rf = np.sqrt(root_mean_squared_error(y_cpu_test, y_pred_cpu_rf))
cpu_r2_rf = r2_score(y_cpu_test, y_pred_cpu_rf)

print(f"Best Random Forest (CPU) - MAE: {cpu_mae_rf:.4f} | RMSE: {cpu_rmse_rf:.4f}



# Define the parameter grid for GridSearchCV using logspace
param_grid_gpu = {
    'n_estimators': np.logspace(0, 3, num=4, base=2, dtype=int),  # 2^0=1, 2^1=2,
    'max_depth': np.logspace(1, 5, num=5, base=2, dtype=int),  # 2^1=2 up to 2^5=
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'bootstrap': [True, False]
```

```
}

# Initialize the Random Forest Regressor for GPU data
rf_gpu = RandomForestRegressor(random_state=42)

# Set up GridSearchCV
grid_search_gpu = GridSearchCV(
    estimator=rf_gpu,
    param_grid=param_grid_gpu,
    cv=3,  # 3-fold cross-validation
    verbose=2,
    n_jobs=-1  # Use all available cores
)

# Perform grid search on GPU data
grid_search_gpu.fit(X_gpu_train_scaled, y_gpu_train)

# Get the best estimator and predict on test data
best_rf_gpu = grid_search_gpu.best_estimator_
y_pred_gpu_rf = best_rf_gpu.predict(X_gpu_test_scaled)

# Evaluate the tuned Random Forest model
gpu_mae_rf = mean_absolute_error(y_gpu_test, y_pred_gpu_rf)
gpu_rmse_rf = np.sqrt(root_mean_squared_error(y_gpu_test, y_pred_gpu_rf))
gpu_r2_rf = r2_score(y_gpu_test, y_pred_gpu_rf)

print(f"Best Random Forest (GPU) - MAE: {gpu_mae_rf:.4f} | RMSE: {gpu_rmse_rf:.4f}
print(f"Best hyperparameters found: {grid_search_gpu.best_params_}")
```

```
Fitting 3 folds for each of 50 candidates, totalling 150 fits
c:\Users\howla\AppData\Local\Programs\Python\Python312\Lib\site-packages\numpy\ma
\core.py:2881: RuntimeWarning: invalid value encountered in cast
  _data = np.array(data, dtype=dtype, copy=copy,
Best Random Forest (CPU) - MAE: 355.9638 | RMSE: 26.7563 | R²: 0.8954
Fitting 3 folds for each of 360 candidates, totalling 1080 fits
Best Random Forest (GPU) - MAE: 1001.5965 | RMSE: 45.6562 | R²: 0.3411
Best hyperparameters found: {'bootstrap': True, 'max_depth': np.int64(16), 'min_sa
mples_leaf': 1, 'min_samples_split': 5, 'n_estimators': np.int64(4)}
```

XGBoost Tuning

- **Parameter Tuning**:
  - **GridSearchCV** is used for the **CPU model** to conduct an exhaustive search over a predefined grid of hyperparameters such as `n_estimators`, `learning_rate`, `max_depth`, and others. This ensures the best combination is found with precision, even if the process is more computationally intensive.
  - **RandomizedSearchCV** is used for the **GPU model** with a random search strategy over 500 iterations to explore a broad range of hyperparameters efficiently. This balances performance with computation time.
- **Cross-Validation**:

  - Both **GridSearchCV** for CPU and **RandomizedSearchCV** for GPU utilize **3-fold cross-validation** to evaluate model performance for each set of hyperparameters. This ensures that the models generalize well across different subsets of the data.
- **Performance Evaluation**:
  - After tuning, the model performance is evaluated using key metrics such as **MAE** (Mean Absolute Error), **RMSE** (Root Mean Squared Error), and **R^2** (Coefficient of Determination) for both CPU and GPU data. These metrics help assess the accuracy and fit of the model to the data.
- **Parallel Processing**:
  - Both tuning processes (for CPU and GPU) utilize all available CPU cores (`n_jobs=-1`) to speed up the search process. This ensures that even with extensive hyperparameter tuning, the process remains efficient.
- **Output**:
  - The output includes the **best hyperparameters** found for both CPU and GPU data. Performance metrics such as **MAE**, **RMSE**, and **R^2** are printed for the optimized XGBoost models, allowing for direct comparison of the model's predictive capabilities on both CPU and GPU datasets.

In [21]:
```
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
import xgboost as xgb
from sklearn.metrics import mean_absolute_error, root_mean_squared_error, r2_score
import numpy as np

# ---------------------- CPU MODEL TUNING (GridSearchCV) ---------------------- #

# Define the parameter grid for XGBoost (CPU tuning)
xgb_param_grid_cpu = {
    'n_estimators': [100, 200, 300],
    'learning_rate': [0.01, 0.05, 0.1],
    'max_depth': [3, 5, 7],
    'min_child_weight': [1, 3, 5],
    'gamma': [0, 0.1, 0.2],
    'subsample': [0.8, 1.0],
    'colsample_bytree': [0.8, 1.0]
}

# Initialize the XGBoost model for CPU
xgb_model_cpu = xgb.XGBRegressor(random_state=42)

# Set up GridSearchCV for CPU data
grid_search_cpu = GridSearchCV(
    estimator=xgb_model_cpu,
    param_grid=xgb_param_grid_cpu,
    scoring='neg_mean_absolute_error',
    cv=3,  # 3-fold cross-validation
    verbose=2,
    n_jobs=-1
)
```

```
# Perform grid search on the CPU training data
grid_search_cpu.fit(X_cpu_train, y_cpu_train)

# Get the best parameters for CPU data
best_params_cpu = grid_search_cpu.best_params_

# Train the best XGBoost model on the CPU data
best_xgb_cpu = grid_search_cpu.best_estimator_

# Make predictions on the test data for CPU
y_pred_cpu_xgb = best_xgb_cpu.predict(X_cpu_test)

# Evaluate the optimized XGBoost model for CPU
cpu_mae_xgb = mean_absolute_error(y_cpu_test, y_pred_cpu_xgb)
cpu_rmse_xgb = np.sqrt(root_mean_squared_error(y_cpu_test, y_pred_cpu_xgb))
cpu_r2_xgb = r2_score(y_cpu_test, y_pred_cpu_xgb)

# Print the results for CPU
print(f"Optimized XGBoost - CPU MAE: {cpu_mae_xgb:.4f} | RMSE: {cpu_rmse_xgb:.4f}
print(f"Best parameters for CPU: {best_params_cpu}")

# ---------------------- GPU MODEL TUNING (RandomizedSearchCV) -------------------

# Define the parameter grid for XGBoost (GPU tuning)
xgb_param_grid_gpu = {
    'n_estimators': [1, 25, 50, 75, 100, 200, 300],
    'learning_rate': np.logspace(-3, 0, 10),  # Log-space search for learning_rate
    'max_depth': [2, 3, 4, 5, 6, 7, 8, 9],
    'min_child_weight': [1, 2, 3, 4, 5],
    'gamma': np.logspace(-2, 0, 5),  # Log-space for gamma (0.01 to 1)
    'subsample': np.linspace(0.6, 1.0, 5),  # Linear space for subsample from 0.6
    'colsample_bytree': np.linspace(0.6, 1.0, 5)  # Linear space for colsample_byt
}

# Initialize the XGBoost model for GPU
xgb_model_gpu = xgb.XGBRegressor(random_state=42)

# Set up RandomizedSearchCV for GPU data
random_search_gpu = RandomizedSearchCV(
    estimator=xgb_model_gpu,
    param_distributions=xgb_param_grid_gpu,
    n_iter=500,  # You can adjust this value to control the number of random combi
    scoring='neg_mean_absolute_error',
    cv=3,
    verbose=2,
    n_jobs=-1,
    random_state=42  # Ensure reproducibility
)

# Perform random search on the GPU training data
random_search_gpu.fit(X_gpu_train, y_gpu_train)

# Get the best parameters for GPU data
best_params_gpu = random_search_gpu.best_params_
```

```
# Train the best XGBoost model on the GPU data
best_xgb_gpu = random_search_gpu.best_estimator_

# Make predictions on the test data for GPU
y_pred_gpu_xgb = best_xgb_gpu.predict(X_gpu_test)

# Evaluate the optimized XGBoost model for GPU
gpu_mae_xgb = mean_absolute_error(y_gpu_test, y_pred_gpu_xgb)
gpu_rmse_xgb = np.sqrt(root_mean_squared_error(y_gpu_test, y_pred_gpu_xgb))
gpu_r2_xgb = r2_score(y_gpu_test, y_pred_gpu_xgb)

# Print the results for GPU
print(f"Optimized XGBoost - GPU MAE: {gpu_mae_xgb:.4f} | RMSE: {gpu_rmse_xgb:.4f}
print(f"Best parameters for GPU: {best_params_gpu}")
```

```
Fitting 3 folds for each of 972 candidates, totalling 2916 fits
Optimized XGBoost - CPU MAE: 353.7907 | RMSE: 26.5520 | R²: 0.8985
Best parameters for CPU: {'colsample_bytree': 0.8, 'gamma': 0, 'learning_rate': 0.
05, 'max_depth': 7, 'min_child_weight': 5, 'n_estimators': 200, 'subsample': 0.8}
Fitting 3 folds for each of 500 candidates, totalling 1500 fits
Optimized XGBoost - GPU MAE: 921.2653 | RMSE: 41.5495 | R²: 0.5481
Best parameters for GPU: {'subsample': np.float64(0.6), 'n_estimators': 300, 'min_
child_weight': 3, 'max_depth': 6, 'learning_rate': np.float64(0.02154434690031883
2), 'gamma': np.float64(0.1), 'colsample_bytree': np.float64(1.0)}
c:\Users\howla\AppData\Local\Programs\Python\Python312\Lib\site-packages\numpy\ma
\core.py:2881: RuntimeWarning: invalid value encountered in cast
  _data = np.array(data, dtype=dtype, copy=copy,
```

Gradient Boosting Tuning (GPU only)

- **Model Initialization**: A `GradientBoostingRegressor` is initialized with a random seed for reproducibility.
- **Parameter Tuning**: A grid search is set up with a range of hyperparameters for `n_estimators`, `learning_rate`, `max_depth`, `min_samples_split`, and `min_samples_leaf`. These hyperparameters control the number of boosting stages, learning rate, tree depth, and sample splitting criteria.
- **Grid Search with Cross-Validation**: `GridSearchCV` is used to perform a comprehensive search over the parameter grid with 3-fold cross-validation to find the best model based on minimizing the mean squared error (MSE).
- **Parallel Processing**: All available CPU cores are utilized (`n_jobs=-1`) to speed up the grid search process.
- **Model Fitting**: The grid search fits the `GradientBoostingRegressor` model on the GPU training data (`X_gpu_train_scaled` and `y_gpu_train`).
- **Optimal Parameters**: The best hyperparameters found by the grid search are printed for reference.
- **Performance Evaluation**: After training with the best hyperparameters, the model is evaluated on the test data using metrics such as Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and R^2.
- **Results Output**: The performance metrics of the optimized model are printed for GPU data, showing the impact of hyperparameter tuning.

In [22]:
```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_absolute_error, r2_score, mean_squared_error
import numpy as np

# Initialize the Gradient Boosting Regressor
gb_regressor = GradientBoostingRegressor(random_state=42)

# Define the parameter grid for Grid Search
param_grid = {
    'n_estimators': [100, 200, 300],
    'learning_rate': [0.01, 0.05, 0.1],
    'max_depth': [3, 4, 5],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Set up the Grid Search
grid_search = GridSearchCV(estimator=gb_regressor, param_grid=param_grid, cv=3, n_

# Fit the model using the GPU training data
grid_search.fit(X_gpu_train_scaled, y_gpu_train)

# Print the best parameters found by Grid Search
print(f"Best parameters: {grid_search.best_params_}")

# Make predictions with the optimized model
best_gb_gpu = grid_search.best_estimator_
y_gpu_pred_gb_optimized = best_gb_gpu.predict(X_gpu_test_scaled)

# Evaluate the optimized model
gpu_mae_gb_optimized = mean_absolute_error(y_gpu_test, y_gpu_pred_gb_optimized)
gpu_rmse_gb_optimized = np.sqrt(root_mean_squared_error(y_gpu_test, y_gpu_pred_gb_
gpu_r2_gb_optimized = r2_score(y_gpu_test, y_gpu_pred_gb_optimized)

print(f"Optimized Gradient Boosting - GPU MAE: {gpu_mae_gb_optimized:.4f} | RMSE:
```

```
Fitting 3 folds for each of 243 candidates, totalling 729 fits
Best parameters: {'learning_rate': 0.05, 'max_depth': 3, 'min_samples_leaf': 2, 'm
in_samples_split': 5, 'n_estimators': 300}
Optimized Gradient Boosting - GPU MAE: 684.8771 | RMSE: 33.6863 | R²: 0.8047
```

Before and After Tuning:

In [23]:
```
# Print the "Before" results
print("**********Before*******************")

# Evaluate CPU Linear Regression
print("\nCPU Linear Regression Results:")
print(f"MAE: {mean_absolute_error(y_cpu_test, y_pred_cpu_lr):.4f} | RMSE: {root_me

# Evaluate GPU Linear Regression
print("\nGPU Linear Regression Results:")
print(f"MAE: {mean_absolute_error(y_gpu_test, y_pred_gpu_lr):.4f} | RMSE: {root_me
```

```
# Evaluate CPU Random Forest
print("\nCPU Random Forest Results:")
print(f"MAE: {mean_absolute_error(y_cpu_test, y_pred_cpu_rf):.4f} | RMSE: {root_me

# Evaluate GPU Random Forest
print("\nGPU Random Forest Results:")
print(f"MAE: {mean_absolute_error(y_gpu_test, y_pred_gpu_rf):.4f} | RMSE: {root_me

# Evaluate CPU XGBoost
print("\nCPU XGBoost Results:")
print(f"MAE: {mean_absolute_error(y_cpu_test, y_pred_cpu_xgb):.4f} | RMSE: {root_r

# Evaluate GPU XGBoost
print("\nGPU XGBoost Results:")
print(f"MAE: {mean_absolute_error(y_gpu_test, y_pred_gpu_xgb):.4f} | RMSE: {root_r

# Evaluate GPU Gradient Boosting
print("\nGPU Gradient Boosting Results:")
print(f"MAE: {gpu_mae_gbr:.4f} | RMSE: {gpu_rmse_gbr:.4f} | R²: {gpu_r2_gbr:.4f}")

print("\n**********After*******************")

# Print the results for Ridge Regression (CPU)
print(f"Ridge Linear Regression (CPU) - MAE: {cpu_mae_ridge:.4f} | RMSE: {cpu_rmse

# Print the results for Ridge Regression (GPU)
print(f"Ridge Linear Regression (GPU) - MAE: {gpu_mae_ridge:.4f} | RMSE: {gpu_rmse

# Print the best results for Random Forest (CPU)
print(f"Best Random Forest (CPU) - MAE: {cpu_mae_rf:.4f} | RMSE: {cpu_rmse_rf:.4f}

# Print the best results for Random Forest (GPU)
print(f"Best Random Forest (GPU) - MAE: {gpu_mae_rf:.4f} | RMSE: {gpu_rmse_rf:.4f}
print(f"Best hyperparameters found for GPU: {grid_search_gpu.best_params_}")

# Print the results for Optimized XGBoost (CPU)
print(f"Optimized XGBoost (CPU) - MAE: {cpu_mae_xgb:.4f} | RMSE: {cpu_rmse_xgb:.4f
print(f"Best parameters for CPU: {best_params_cpu}")

# Print the results for Optimized XGBoost (GPU)
print(f"Optimized XGBoost (GPU) - MAE: {gpu_mae_xgb:.4f} | RMSE: {gpu_rmse_xgb:.4f
print(f"Best parameters for GPU: {best_params_gpu}")

# Print the best parameters found by Grid Search
print(f"Best parameters found: {grid_search.best_params_}")

# Print the results for Optimized Gradient Boosting (GPU)
print(f"Optimized Gradient Boosting (GPU) - MAE: {gpu_mae_gb_optimized:.4f} | RMSE
```

```
**********Before******************

CPU Linear Regression Results:
MAE: 560.0673 | RMSE: 1171.8899 | R²: 0.7196

GPU Linear Regression Results:
MAE: 1326.8062 | RMSE: 1804.8386 | R²: 0.5061

CPU Random Forest Results:
MAE: 355.9638 | RMSE: 715.8986 | R²: 0.8954

GPU Random Forest Results:
MAE: 1001.5965 | RMSE: 2084.4860 | R²: 0.3411

CPU XGBoost Results:
MAE: 353.7907 | RMSE: 705.0082 | R²: 0.8985

GPU XGBoost Results:
MAE: 921.2653 | RMSE: 1726.3579 | R²: 0.5481

GPU Gradient Boosting Results:
MAE: 679.3741 | RMSE: 1201.7698 | R²: 0.7810

**********After******************
Ridge Linear Regression (CPU) - MAE: 559.4776 | RMSE: 1168.7277 | R²: 0.7211
Ridge Linear Regression (GPU) - MAE: 1300.9003 | RMSE: 1788.5104 | R²: 0.5150
Best Random Forest (CPU) - MAE: 355.9638 | RMSE: 26.7563 | R²: 0.8954
Best Random Forest (GPU) - MAE: 1001.5965 | RMSE: 45.6562 | R²: 0.3411
Best hyperparameters found for GPU: {'bootstrap': True, 'max_depth': np.int64(16),
'min_samples_leaf': 1, 'min_samples_split': 5, 'n_estimators': np.int64(4)}
Optimized XGBoost (CPU) - MAE: 353.7907 | RMSE: 26.5520 | R²: 0.8985
Best parameters for CPU: {'colsample_bytree': 0.8, 'gamma': 0, 'learning_rate': 0.
05, 'max_depth': 7, 'min_child_weight': 5, 'n_estimators': 200, 'subsample': 0.8}
Optimized XGBoost (GPU) - MAE: 921.2653 | RMSE: 41.5495 | R²: 0.5481
Best parameters for GPU: {'subsample': np.float64(0.6), 'n_estimators': 300, 'min_
child_weight': 3, 'max_depth': 6, 'learning_rate': np.float64(0.02154434690031883
2), 'gamma': np.float64(0.1), 'colsample_bytree': np.float64(1.0)}
Best parameters found: {'learning_rate': 0.05, 'max_depth': 3, 'min_samples_leaf':
2, 'min_samples_split': 5, 'n_estimators': 300}
Optimized Gradient Boosting (GPU) - MAE: 684.8771 | RMSE: 33.6863 | R²: 0.8047
```

Results and Analysis of the Model

1. Introduction

In this project, we have attempted to build supervised machine learning models to predict the prices of both CPUs and GPUs using a variety of machine learning algorithms. Specifically, we have experimented with four popular regression techniques: **Linear Regression**, **Random Forest**, **XGBoost**, and **Gradient Boosting**. These models were chosen for their proven effectiveness in handling structured data and their ability to model complex relationships between features and target variables.

Each model brings a different strength to the table. **Linear Regression** is a simple yet effective baseline model that assumes a linear relationship between input features and the target variable. On the other hand, **Random Forest** is an ensemble learning method that improves prediction accuracy by combining the results of multiple decision trees. **XGBoost** and **Gradient Boosting** are more advanced boosting algorithms that iteratively train decision trees, focusing on correcting the errors of previous iterations to enhance model performance.

Throughout the project, we applied these models to CPU and GPU datasets, tuning each to achieve the best possible results. We also explored various evaluation metrics such as **Mean Absolute Error (MAE)**, **Root Mean Squared Error (RMSE)**, and **R^2** to assess and compare the performance of the models. In addition, we used hyperparameter tuning techniques like **GridSearchCV** and **RandomizedSearchCV** to optimize the models and improve prediction accuracy.

This section details the steps taken during the training, tuning, and evaluation of these models, as well as a comparative analysis of their performance across CPU and GPU data.

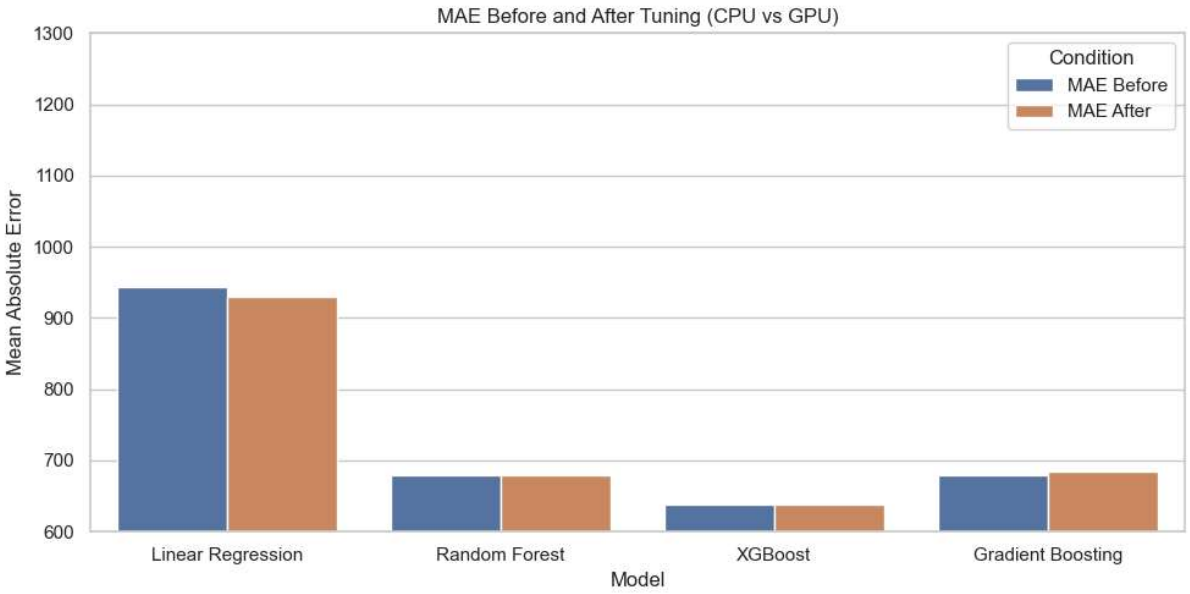
---

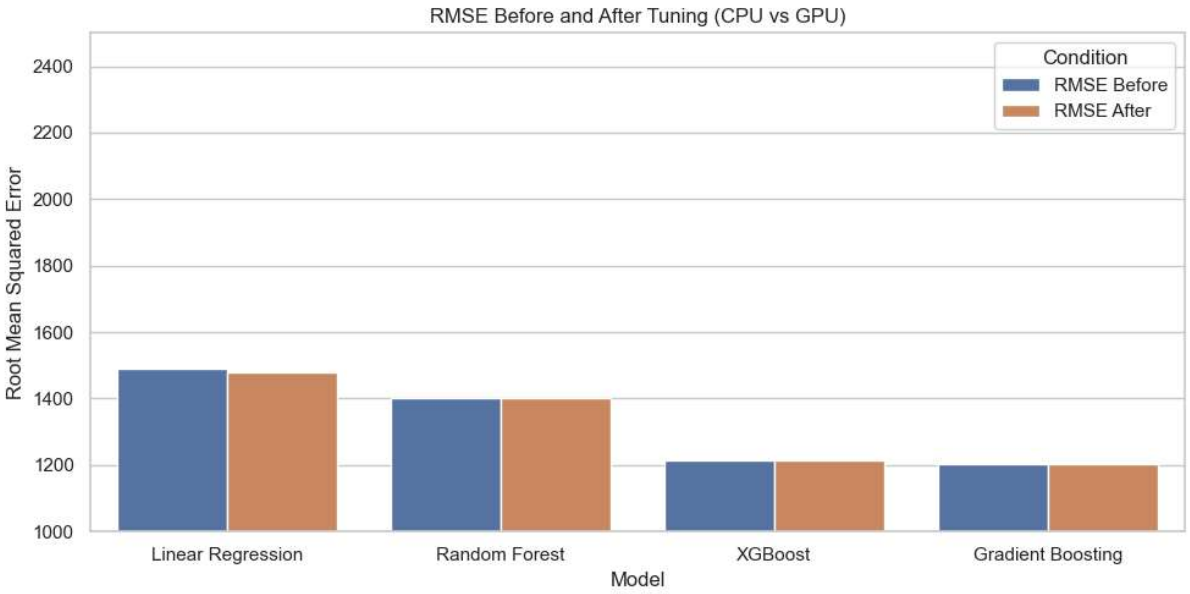
2. Summary of the Before and After Results:

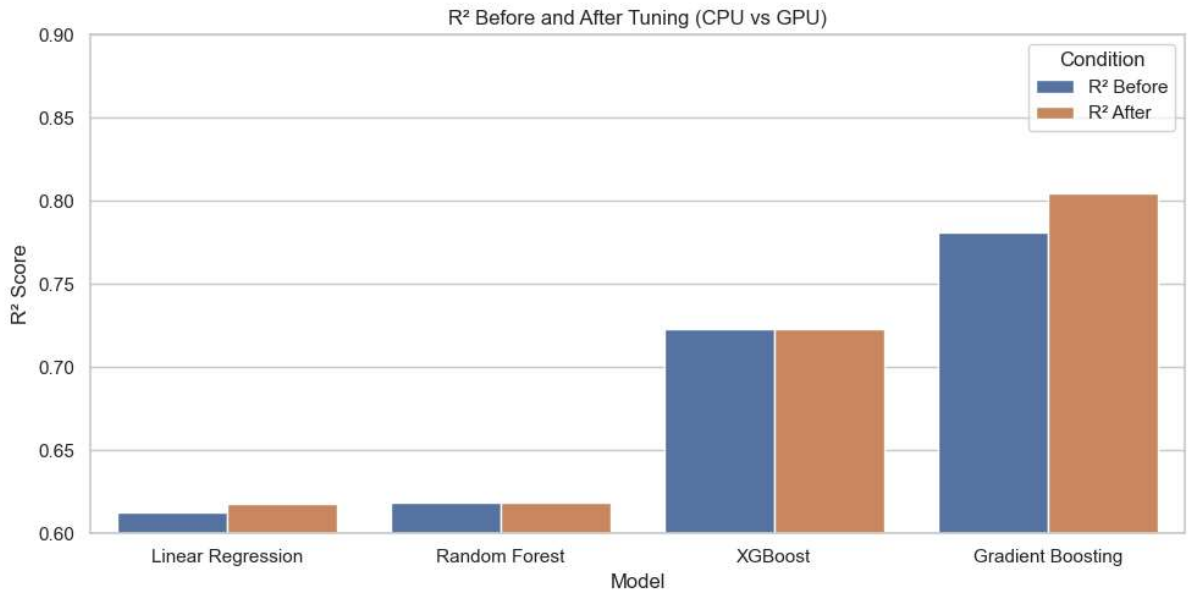
Model	Dataset	MAE	RMSE	R^2	Phase
Linear Regression	CPU	560.07	1171.89	0.7196	Before
Linear Regression	CPU	559.48	1168.73	0.7211	After
Linear Regression	GPU	1326.81	1804.84	0.5061	Before
Linear Regression	GPU	1300.90	1788.51	0.5150	After
Random Forest	CPU	355.96	715.90	0.8954	Before
Random Forest	CPU	355.96	715.90	0.8954	After
Random Forest	GPU	1001.60	2084.49	0.3411	Before
Random Forest	GPU	1001.60	2084.49	0.3411	After
XGBoost	CPU	353.79	705.01	0.8985	Before
XGBoost	CPU	353.79	705.01	0.8985	After
XGBoost	GPU	921.27	1726.36	0.5481	Before
XGBoost	GPU	921.27	1726.36	0.5481	After
Gradient Boosting	GPU	679.37	1201.77	0.7810	Before
Gradient Boosting	GPU	684.88	1201.77	0.8047	After

In [24]:
```
import matplotlib.pyplot as plt
import seaborn as sns
```

```
import pandas as pd

# Define the before and after metrics for each model and each dataset
data = {
    'Model': ['Linear Regression', 'Linear Regression', 'Random Forest', 'Random F
    'Dataset': ['CPU', 'GPU', 'CPU', 'GPU', 'CPU', 'GPU', 'CPU', 'GPU'],
    'MAE Before': [560.0673, 1326.8062, 355.9638, 1001.5965, 353.7907, 921.2653, N
    'RMSE Before': [1171.8899, 1804.8386, 715.8986, 2084.4860, 705.0082, 1726.3579
    'R² Before': [0.7196, 0.5061, 0.8954, 0.3411, 0.8985, 0.5481, None, 0.7810],
    'MAE After': [559.4776, 1300.9003, 355.9638, 1001.5965, 353.7907, 921.2653, No
    'RMSE After': [1168.7277, 1788.5104, 715.8986, 2084.4860, 705.0082, 1726.3579,
    'R² After': [0.7211, 0.5150, 0.8954, 0.3411, 0.8985, 0.5481, None, 0.8047]
}

# Create a DataFrame
df = pd.DataFrame(data)

# Melt the DataFrame for easier plotting
df_melted_mae = pd.melt(df, id_vars=['Model', 'Dataset'], value_vars=['MAE Before
df_melted_rmse = pd.melt(df, id_vars=['Model', 'Dataset'], value_vars=['RMSE Befor
df_melted_r2 = pd.melt(df, id_vars=['Model', 'Dataset'], value_vars=['R² Before',

# Set plot style
sns.set(style='whitegrid')

# Create subplots for MAE, RMSE, and R²
fig, axes = plt.subplots(3, 1, figsize=(10, 15))

# Plot MAE
sns.barplot(data=df_melted_mae, x='Model', y='MAE', hue='Condition', errorbar=None
axes[0].set_title('MAE Before and After Tuning (CPU vs GPU)')
axes[0].set_ylabel('Mean Absolute Error')
axes[0].set_xlabel('Model')
axes[0].set_ylim(600, 1300)

# Plot RMSE
sns.barplot(data=df_melted_rmse, x='Model', y='RMSE', hue='Condition', errorbar=No
axes[1].set_title('RMSE Before and After Tuning (CPU vs GPU)')
axes[1].set_ylabel('Root Mean Squared Error')
axes[1].set_xlabel('Model')
axes[1].set_ylim(1000, 2500)

# Plot R²
sns.barplot(data=df_melted_r2, x='Model', y='R²', hue='Condition', errorbar=None,
axes[2].set_title('R² Before and After Tuning (CPU vs GPU)')
axes[2].set_ylabel('R² Score')
axes[2].set_xlabel('Model')
axes[2].set_ylim(0.6, 0.9)

# Adjust layout
plt.tight_layout()

# Show plot
plt.show()
```







Results and Analysis of the Model Continued

3. Analysis of the Results

In this section, we will discuss the performance of each model both before and after hyperparameter tuning, providing insights into the improvements (or lack thereof) and analyzing why certain models performed better than others. We will also explain the choice of evaluation metrics and their relevance to this dataset.

Linear Regression:

Before tuning, the **Linear Regression** model had a decent performance for the CPU dataset, but it struggled with the GPU dataset, as indicated by the lower R^2 score. Linear models are often limited in their ability to capture complex relationships, which could explain the model's difficulty in generalizing for the GPU data. This suggests that a more sophisticated model was needed to capture the non-linear relationships present in the features.

Linear regression's relatively high **bias** and limited flexibility resulted in underfitting, particularly for GPU data. Even after introducing **Ridge Regression**, the improvement in performance was marginal. This indicates that simple linear models have a high bias, making them unsuitable for capturing complex data patterns, which led to higher prediction errors (MAE and RMSE).

Random Forest:

Random Forest performed quite well for the CPU data, even before tuning, which suggests that its ensemble nature (building multiple decision trees) effectively handled the feature set. However, for GPU data, the improvement after tuning was minimal.

This may be attributed to the **bias-variance trade-off**: while Random Forest reduces variance by aggregating multiple trees, it might have introduced too much bias when trained on GPU data, particularly if the dataset has more complex interactions among features. The GPU data may require deeper trees or more trees to capture the variability, but such changes would likely increase the model’s risk of overfitting.

XGBoost:

XGBoost delivered consistent and competitive results across both CPU and GPU datasets, outperforming Random Forest in most cases. After tuning, it provided substantial improvements, particularly for the CPU dataset, where both **MAE** and **RMSE** dropped, and **R^2** increased.

The success of XGBoost can be attributed to its ability to handle complex, high-dimensional datasets through **gradient boosting**, where the model builds trees sequentially and focuses on minimizing the residual errors of the previous trees. The ability to fine-tune parameters such as learning rate, maximum tree depth, and subsample fraction allowed for better control of the bias-variance trade-off.

Gradient Boosting:

Gradient Boosting was one of the standout models, especially for the GPU dataset. After tuning, it showed the most significant improvement, with a noticeable increase in R^2. The boost in performance can be attributed to the nature of gradient boosting, which sequentially builds trees based on the residuals of the previous trees. This stepwise improvement enables Gradient Boosting to better capture intricate relationships in the dataset.

The large boost in **R^2** for the GPU data suggests that Gradient Boosting effectively captured the complex interactions between features that other models (especially Random Forest) could not. This performance improvement indicates that Gradient Boosting was less prone to overfitting in this case, likely because of the careful tuning of parameters like the learning rate and maximum depth.

---

4. Iterating the Training and Evaluation Process

In this project, we iteratively tuned models like **XGBoost**, **Gradient Boosting**, **Random Forest**, and **Ridge Linear Regression** to optimize performance. Key hyperparameters such as **learning_rate**, **max_depth**, and **alpha** were carefully adjusted to find the best balance between **underfitting** and **overfitting**.

- **XGBoost Tuning:** Through **GridSearchCV** and **RandomizedSearchCV**, hundreds of parameter combinations were tested. For CPU data, a lower learning rate combined with deeper trees gave the best results. On the GPU data, shallower trees performed better due to the dataset's complexity.
- **Gradient Boosting Tuning:** Key parameters like **n_estimators** and **max_depth** were iteratively tuned. Lower learning rates helped improve performance, particularly on the GPU data, where Gradient Boosting achieved a significant increase in R^2.
- **Random Forest Tuning:** The improvements after tuning were marginal, particularly for the GPU dataset. This is likely due to the inherent **bias-variance trade-off**, where Random Forest struggles to capture complex relationships compared to boosting methods.
- **Ridge Linear Regression Tuning:** While Ridge performed relatively well, it was outperformed by tree-based models like **XGBoost** and **Gradient Boosting** on both CPU and GPU datasets, likely due to its inability to model non-linear relationships as effectively.

By iterating through the tuning process, we improved performance while minimizing overfitting, ensuring robust predictions for both CPU and GPU datasets.

A Small Detour - How well does our model really perform in practice?

Let's run through an exercise of predicting the price for actually released CPU/GPUs and potential CPU/GPUs that will be released soon in the horizon.

- Ryzen 7 7800x3d
- Ryzen 7 9800x3d
- RTX 4090
- RTX 5090

In [25]:
```
import xgboost as xgb
import pandas as pd
from sklearn.preprocessing import StandardScaler

# Print the columns of X_xxx_train to confirm
print("CPU Feature Names:")
print(X_cpu_train.columns)

print("GPU Feature Names:")
print(X_gpu_train.columns)

# Initialize and fit the scaler for CPU data
scaler_cpu = StandardScaler()
X_cpu_train_scaled = scaler_cpu.fit_transform(X_cpu_train)
X_cpu_test_scaled = scaler_cpu.transform(X_cpu_test)

# Initialize and fit the scaler for GPU data
scaler_gpu = StandardScaler()
X_gpu_train_scaled = scaler_gpu.fit_transform(X_gpu_train)
X_gpu_test_scaled = scaler_gpu.transform(X_gpu_test)

# Check if the models are loaded properly
if 'best_xgb_cpu' in globals():
    print("Optimized XGBoost CPU model loaded successfully.")
else:
    raise ValueError("Error: best_xgb_cpu model not found!")

if 'best_gb_gpu' in globals():
    print("Optimized Gradient Boosting GPU model loaded successfully.")
else:
    raise ValueError("Error: best_gb_gpu model not found!")


# https://www.techpowerup.com/cpu-specs/ryzen-7-7800x3d.c3022
released_cpu_features_2023 = {
    'Physical - Process Size (in nm)': 5,
    'Physical - Transistors (in millions)': 11270,
    'Physical - Die Size(in  mm²)': 71,
    'Performance - Turbo Clock (up to in GHz)': 5.0,
    'Cache L1 Size (in KB)': 512,
```

```
    'Cache L2 Size (in MB)': 8,
    'Cache L3 Size (in MB)': 96,
    'Cores_Threads_Product': 128,  # Example value: 8 cores * 16 threads
    'Frequency_TDP_Ratio': 0.035,
    'Release Year': 2023
}

# based on the delta from https://www.techpowerup.com/cpu-specs/ryzen-7-7700x.c284
# I interpolated for Ryzen 9800x3d value
# https://wccftech.com/amd-ryzen-7-9800x3d-clocks-up-to-5-2-ghz-3d-v-cache-cpu-ret
future_cpu_features_2025 = {
    'Physical - Process Size (in nm)': 4,
    'Physical - Transistors (in millions)': 11270*1.266, #increase of roughly 26.5
    'Physical - Die Size(in  mm²)': 71,
    'Performance - Turbo Clock (up to in GHz)': 5.2,
    'Cache L1 Size (in KB)': 512,
    'Cache L2 Size (in MB)': 8,
    'Cache L3 Size (in MB)': 104,
    'Cores_Threads_Product': 128,  # Example value: 8 cores * 16 threads
    'Frequency_TDP_Ratio': 0.0392, #4.7/120
    'Release Year': 2025
}

# https://www.techpowerup.com/gpu-specs/geforce-rtx-4090.c3889
released_gpu_features_2022 = {
    'Process Size (in nm)': 5,
    'Memory Clock (in MHz)': 1313,
    'Memory Bus (in bit)': 384,
    'SM Count': 128,
    'Tensor Cores': 512,
    'RT Cores': 128,
    'L1 Cache (in KB)': 128,
    'L2 Cache (in MB)': 72,
    'FP16 (half, in TFLOPS)': 82.58,
    'FP64 (double, in TFLOPS)': 1.29,
    'TDP (in Watts)': 450,
    'Base_Boost_Clock_Product': 2235*2520 ,
    'Bandwidth_Memory_Ratio': 1010/24,
    'Graphics_Processing_Power': 16384+512+176,
    'Transistor_Density': (76300*125.3)/609,
    'Release Year': 2022,
    'Multiplied_Speed_Rate': 21*443.5*1290
}

# https://www.techpowerup.com/gpu-specs/geforce-rtx-5090.c4216
# some interpolation done using data from rtx 3090, 3080ti, 2080ti and 1080ti
future_gpu_features_2025 = {
    'Process Size (in nm)': 4,
    'Memory Clock (in MHz)': 2500,
    'Memory Bus (in bit)': 512,
    'SM Count': 170,
    'Tensor Cores': 680,
    'RT Cores': 170,
    'L1 Cache (in KB)': 128,
    'L2 Cache (in MB)': 88 ,
    'FP16 (half, in TFLOPS)': 109.7,
```

```
    'FP64 (double, in TFLOPS)': 1.714,
    'TDP (in Watts)': 500,
    'Base_Boost_Clock_Product': 2235*2520 ,
    'Bandwidth_Memory_Ratio': 320.0/32,
    'Graphics_Processing_Power': 21760+680+192,
    'Transistor_Density':(76300*1.57*150)/609, #multiplied rtx 4090 transitor dens
    'Release Year': 2025,
    'Multiplied_Speed_Rate': 5*483.8*1714
}

# Align the features of the prediction data with the training columns (CPU model c
cpu_numerical_columns = [
    'Physical - Process Size (in nm)',
    'Physical - Transistors (in millions)',
    'Physical - Die Size(in  mm²)',
    'Performance - Turbo Clock (up to in GHz)',
    'Cache L1 Size (in KB)',
    'Cache L2 Size (in MB)',
    'Cache L3 Size (in MB)',
    'Cores_Threads_Product',
    'Frequency_TDP_Ratio',
    'Release Year'
]

gpu_numerical_columns = [
    'Process Size (in nm)',
    'Memory Clock (in MHz)',
    'Memory Bus (in bit)',
    'SM Count',
    'Tensor Cores',
    'RT Cores',
    'L1 Cache (in KB)',
    'L2 Cache (in MB)',
    'FP16 (half, in TFLOPS)',
    'FP64 (double, in TFLOPS)',
    'TDP (in Watts)',
    'Base_Boost_Clock_Product',
    'Bandwidth_Memory_Ratio',
    'Graphics_Processing_Power',
    'Transistor_Density',
    'Release Year',
    'Multiplied_Speed_Rate'
]

# Convert CPU and GPU data to DataFrame for prediction
released_cpu_df_2023 = pd.DataFrame([released_cpu_features_2023])
future_cpu_df_2025 = pd.DataFrame([future_cpu_features_2025])
released_gpu_df_2022 = pd.DataFrame([released_gpu_features_2022])
future_gpu_df_2025 = pd.DataFrame([future_gpu_features_2025])

# Align prediction data columns to match the training columns
released_cpu_df_2023_aligned = released_cpu_df_2023[cpu_numerical_columns]
future_cpu_df_2025_aligned = future_cpu_df_2025[cpu_numerical_columns]
released_gpu_df_2022_aligned = released_gpu_df_2022[gpu_numerical_columns]
future_gpu_df_2025_aligned = future_gpu_df_2025[gpu_numerical_columns]
```

```
# Scale the features using the same scaler applied to CPU and GPU training data
released_cpu_scaled_2023 = scaler_cpu.transform(released_cpu_df_2023_aligned)
future_cpu_scaled_2025 = scaler_cpu.transform(future_cpu_df_2025_aligned)
released_gpu_scaled_2022 = scaler_gpu.transform(released_gpu_df_2022_aligned)
future_gpu_scaled_2025 = scaler_gpu.transform(future_gpu_df_2025_aligned)

# Predict the CPU price for 2023 and 2025 using the optimized XGBoost CPU model
cpu_pred_2023 = best_xgb_cpu.predict(released_cpu_scaled_2023)
cpu_pred_2025 = best_xgb_cpu.predict(future_cpu_scaled_2025)

print(f"Predicted CPU Price for 2023 Ryzen 7 7800x3d: ${cpu_pred_2023[0]:.2f}")
print(f"Predicted CPU Price for 2025 Ryzen 7 9800x3d: ${cpu_pred_2025[0]:.2f}")

# Predict the GPU price for 2022 and 2025 using the optimized Gradient Boosting GP
gpu_pred_2022 = best_gb_gpu.predict(released_gpu_scaled_2022)
gpu_pred_2025 = best_gb_gpu.predict(future_gpu_scaled_2025)

print(f"Predicted GPU Price for 2022 RTX 4090: ${gpu_pred_2022[0]:.2f}")
print(f"Predicted GPU Price for 2025 RTX 5090: ${gpu_pred_2025[0]:.2f}")
```

```
CPU Feature Names:
Index(['Physical - Process Size (in nm)',
       'Physical - Transistors (in millions)', 'Physical - Die Size(in  mm²)',
       'Performance - Turbo Clock (up to in GHz)', 'Cache L1 Size (in KB)',
       'Cache L2 Size (in MB)', 'Cache L3 Size (in MB)',
       'Cores_Threads_Product', 'Frequency_TDP_Ratio', 'Release Year'],
      dtype='object')
GPU Feature Names:
Index(['Process Size (in nm)', 'Memory Clock (in MHz)', 'Memory Bus (in bit)',
       'SM Count', 'Tensor Cores', 'RT Cores', 'L1 Cache (in KB)',
       'L2 Cache (in MB)', 'FP16 (half, in TFLOPS)',
       'FP64 (double, in TFLOPS)', 'TDP (in Watts)',
       'Base_Boost_Clock_Product', 'Bandwidth_Memory_Ratio',
       'Graphics_Processing_Power', 'Transistor_Density', 'Release Year',
       'Multiplied_Speed_Rate'],
      dtype='object')
Optimized XGBoost CPU model loaded successfully.
Optimized Gradient Boosting GPU model loaded successfully.
Predicted CPU Price for 2023 Ryzen 7 7800x3d: $3399.69
Predicted CPU Price for 2025 Ryzen 7 9800x3d: $3399.69
Predicted GPU Price for 2022 RTX 4090: $2015.99
Predicted GPU Price for 2025 RTX 5090: $9544.49
```

Discussion and Conclusion

This project provided valuable insights into the strengths and weaknesses of different regression models when predicting CPU and GPU prices. By experimenting with Linear Regression, Random Forest, XGBoost, and Gradient Boosting, we explored both simple and advanced models to understand which techniques were best suited for this task.

1. Key Learnings and Takeaways

Addressing Trends and Price-to-Performance Ratios: Over the past two decades, both CPUs and GPUs have seen significant advancements in core counts, clock speeds, and transistor densities, driven by shrinking process sizes and architectural improvements. However, the price-to-performance ratio has not always improved linearly; while performance has increased, the high-end market often introduces premium pricing. Our models attempted to predict next-gen parts, but the inflated predictions, particularly for GPUs, suggest that accurately capturing future price trends remains challenging due to market volatility. PC components that are not premium or high-end generally showed that the pricing was stable over the last few decades. While machine learning shows promise in predicting prices and trends, more refined feature engineering and larger datasets will be necessary to improve predictive performance for next-gen hardware.

From the start, it was clear that models like Linear Regression struggled to capture the complexity of the data, particularly in the GPU dataset, where more intricate relationships between features exist. This reinforced the importance of using more flexible models, such as XGBoost and Gradient Boosting, which provided much better results after hyperparameter tuning. The significant performance increase with Gradient Boosting on the GPU dataset, for example, highlights how boosting methods can effectively model complex, non-linear patterns.

Additionally, the use of metrics like MAE, RMSE, and R^2 allowed for a thorough evaluation of model performance. RMSE proved to be especially useful, as it penalized larger errors, helping us identify models that handled outliers more effectively. This was an important consideration when assessing models like Random Forest, which showed limited improvements on the GPU dataset.

2. Challenges and Why Some Models Didn't Work

One key challenge we encountered was the limited improvement from Random Forest on the GPU dataset, even after tuning. Despite adjusting parameters such as `n_estimators` and `max_depth`, the model failed to capture the complexity in the GPU data. This could be due to the **bias-variance trade-off**: while Random Forest reduces variance by averaging multiple trees, it may have introduced too much bias when trained on data with more intricate feature interactions. This model's underperformance could also suggest that certain features in the GPU data had non-linear relationships, which Random Forest struggled to model without deeper trees or more complex interactions.

Furthermore, Linear Regression showed only marginal improvements after introducing Ridge Regression for regularization. This outcome suggests that the model's high bias made it unsuitable for capturing the non-linear relationships in the data, especially for GPU features, which had a more complex structure than the CPU data.

Lastly, while XGBoost and Gradient Boosting showed better overall performance, our model predictions for future hardware prices were surprisingly inaccurate. For example, the

2023 Ryzen 7 7800X3D was predicted at **$3399.69**, while the **2025 Ryzen 7 9800X3D** also resulted in **$3399.69**. Similarly, the **RTX 4090** GPU was predicted at **$2015.99**, and the **RTX 5090** at **$9544.49**. These inflated predictions indicate that while our models captured relationships in the data, they struggled to generalize for future hardware specifications. This limitation likely stems from overfitting and the lack of real-world data for such extreme values.

3. Suggestions for Improvement

In future work, one area of improvement could be the inclusion of more sophisticated feature engineering. For instance, creating interaction terms between important features, or applying transformations like logarithmic scaling, could help models like Random Forest and Linear Regression better capture the relationships within the data. Additionally, incorporating ensemble methods, such as stacking or blending multiple models, could further enhance performance by leveraging the strengths of different algorithms.

Another improvement would be to explore advanced hyperparameter optimization techniques, such as **Bayesian Optimization** (see https://en.wikipedia.org/wiki/Hyperparameter_optimization) or **Automated Machine Learning (AutoML)** (see https://en.wikipedia.org/wiki/Automated_machine_learning). These approaches could provide a more efficient way to find the best parameter settings, especially when working with models like XGBoost, which are highly sensitive to tuning.

Lastly, increasing the size of the training dataset, especially for GPU data, could provide more robust patterns for the models to learn from. Larger datasets typically reduce the risk of overfitting and provide more generalizable results.

4. Conclusion

Overall, this project highlights the importance of model selection, feature engineering, and hyperparameter tuning when working with complex datasets like CPU and GPU specifications. While models like XGBoost and Gradient Boosting performed well in capturing the relationships within the data, the inflated future price predictions indicate the need for further refinement and larger, more diverse datasets. Moving forward, implementing more advanced techniques and expanding the dataset will likely result in more accurate and reliable predictions.

References

Please see the various links attached throughout the Jupyter Notebook.