Automated DBMS Fuzzing Framework

Mid-Term Progress Report

CSCE 555, Spring ‘25, UL

*Rakshya Pandey(C00580017), Surendar Kumar(C00577830)*

March 30, 2024

# Introduction

Nowadays, there is a huge advancement in deep learning models ranging from computer vision to natural language processing. The efficiency of these deep learning models is not just defined based on the effectiveness of the algorithms but is also dependent on underlying hardware requirements that are used during the training phase. Deep learning models heavily rely on complex mathematical operations, and some of the core mathematical operations are suitable to be parallelized because parallel operation increases the execution speed.

For any machine learning or deep learning model, there are two processing units that are extend- sively used: the Central Processing Unit (CPU) and the Graphical Processing Unit (GPU). CPUs are known for their versatility and ability to handle general-purpose computing tasks. CPUs are excellent at working with sequential tasks in an efficient and quick manner. They are not optimized to work parallel processing tasks. In contrast, GPUs are specifically designed to work in parallel. The normal CPUs typically have 4-5 cores and only a limited number of threads can be handled. GPUs have thousands of small cores capable of executing operations simultaneously, and GPUs demonstrate exceptional power in parallel processing tasks. Thus, the parallel processing capable- it of the GPU is higher than that of the CPU.

It is important to understand the capabilities and functionalities of both CPUs and GPUs while training a deep learning model because this understanding helps to make informed decisions when selecting hardware and allocating resources. Profiling is a technique to measure the program’s resource utilization, execution time, and performance characteristics.

# Related Works

There are a number of previous researches conducted to identify the performances of CPU and GPU while performing deep learning tasks. For instance, Buber E. and Banu D.’s study [[2](#_bookmark1)] high- light the performance analysis of deep learning applications in classifying web pages, emphasizing comparisons between CPU and GPU servers in cloud environments. The findings of the paper indicate that GPU servers consistently outperformed CPU servers, achieving acceleration rates up to 4-5 times faster in various scenarios. Lind E. and Pantigoso V. [[3](#_bookmark2)] talked about the impact of

processing units on TensorFlow’s performance where he found that GPUs are advantageous for complex networks. However, simpler models with limited data show minimal GPU benefits. Mem- ory management comparisons between GPUs and CPUs remain inconclusive, primarily affected by training data size.

Additionally, Sankara et al. [[6](#_bookmark5)] presented a comparison of the performance between CPU and GPU by running DNN models using TensorFlow. The study found that GPUs (Tesla k20c and Titan X) outperformed CPUs ranging in speed increment from 40x and 82x while running the inceptionV3 model. Similarly, for the ResNet50 model, the speedup relative to CPU was 52x for Tesla k20c and 102x for Titan X GPUs. The paper by Voon et al. [[8](#_bookmark7)] does the comparison between CPU, GPU, and TPU’s performance on image classification with the VGG16 model for classifying images on CIFAR-10 and MNIST datasets. It was observed that the GPU was able to execute 25x faster than the CPU and the TPU was 12x faster than the CPU. Study done by Munanday et al. [[4](#_bookmark3)] shows that GPU is faster and more efficient compared to CPU and TPU for face emotion recognition using a convolution neural network. This paper highlights the importance of speed and accuracy in deep learning models and shows how having underlying GPU support provides such enhancements.

# Problem and Solution

The primary motivation of this research project is to understand the underlying operations on CPUs and GPUs while working on high computational machine learning and deep learning models. Different machine learning and deep learning models have high mathematical computations and take a lot of time and resources to complete the training of the model. It creates several perfor- mance bottlenecks if we cannot choose appropriate hardware resources for specified modeling tasks.

Analyzing time and memory consumption at runtime will be helpful in pinpointing bottlenecks and areas of inefficiency in the code that contribute to high CPU/GPU utilization. Additionally, it is also possible to find out if there are any parameters of the neural network model responsible for resource consumption across CPU and GPU.

Software profiling can be used to optimize machine learning and deep learning models by know- ing the information related to memory usage, resource utilization, and time taken. Profiling tech- niques like Nvidia-smi, Deep Learning Profiler (DLProf), PyTorch, and PyProf are widely used to know about resource utilization. Nvidia-smi is a Linux command that can be used to get in- formation on memory usage, power consumption, and processes running on GPU. We can analyze whether the GPU is being properly utilized through this command. Deep Learning Profiler is a tool for profiling deep learning models to help data scientists understand and improve the performance of their models visually via the DLProf Viewer or by analyzing text reports [[1](#_bookmark0)]. Similarly, PyTorch Profiler is a tool that allows the collection of performance metrics during training and inference [[5](#_bookmark4)]. These approaches provide researchers with a powerful tool to get deeper insights into the time for execution, memory utilization, and resource consumption by the underlying CPU and GPU. The information obtained from these profiles can be used to compare the performance of the CPU and GPU in the training of the deep learning model.

# Summary of the project

For this project, we are focusing on understanding the performance differences between CPU and GPU while training deep learning models. For the implementation purpose, we will be using profiling approaches to understand in depth usage of the resources. In this project, we will be evaluating a model by custom training with YOLO v8. This study can later be extended for learning and evaluation by performing the test in other models as well.

# Methodology

The project experimentation involves training a custom pre-trained YOLOv8 model for accurate decisions of individuals wearing face masks using the Face Mask Dataset.

* 1. Dataset Selection: The Face Mask Dataset is downloaded from Kaggle, which is an open- source platform.
     1. The dataset comprises a total of 1420 images.
     2. The images are divided into three subsets for training, validation, and testing:
     3. Training set: Consists of 990 images, which accounts for 70% of the dataset.
     4. Validation set: Consists of 294 images, which accounts for 20% of the dataset.
     5. Test set: Consists of 136 images, which accounts for 10% of the dataset.
  2. Model Selection: The YOLO v8 is a cutting-edge, state-of-the-art (SOTA) model that is designed to be fast, accurate, and easy to use, making it an excellent choice for a wide range of object detection and tracking, instance segmentation, image classification, and pose estimation tasks [[7](#_bookmark6)]. This model is efficient for evaluating the performance differences between CPU and GPU.
  3. Model is trained using the Face Mask Dataset. Different numbers of hyperparameters were configured to experiment. We experimented with the batch size, number of epochs, optimizers, and input image size.
  4. Profiling technologies like Nvidia-sim, DlProf, and PyTorch will be used in our training pipeline to evaluate the performance bottlenecks, resource usage, and comparison with respect to CPU.
  5. Profiling data will be analyzed to identify bottlenecks, and a visual representation of the performance differences between CPU and GPU will be shown.

# Current Project Progress

We have made significant progress in our project. We have completed our preliminary reading of the relevant papers, which provided us with a solid theoretical foundation for working on our project. We prepared a dataset configuration file and have completed training our model in Google Colab. While training our model, we experimented with different epoch numbers and batch sizes. Additionally, we are actively learning about different profilers like Pytorch profiler, DlProf on tensorboard, and Nvidia-smi to analyze the usage of CPU and GPU and to understand resource utilization for deep learning tasks.

# Expected Delivery

We aim to deliver a concise yet insightful comparison between CPU and GPU profiling in deep learning models through comparative analysis. We will analyze key metrics like GPU utilization, power utilization, GPU memory access and utilization, and throughput and present them visually. This comparison and graphical representation of the above-mentioned metrics will be helpful in knowing about the nuanced performance differences between the two processing units, facilitating a clear understanding for developers, system architects, and researchers.

# References

1. Deep learning profiler user guide, accessed 2024.
2. Ebubekir Buber and DIRI Banu. Performance analysis and cpu vs gpu comparison for deep learning. In *2018 6th International Conference on Control Engineering & Information Technol- ogy (CEIT)*, pages 1–6. IEEE, 2018.
3. Eric Lind and A¨ velin Pantigoso Velasquez. A performance comparison between cpu and gpu in tensorflow, 2019.
4. Anbananthan Pillai Munanday, Norazlianie Sazali, Wan Sharuzi Wan Harun, Kumaran Kadirgama, and Ahmad Shahir Jamaludin. Analysis of convolutional neural networks for facial expression recognition on gpu, tpu and cpu. *Journal of Advanced Research in Applied Sciences and Engineering Technology*, 31(3):50–67, 2023.
5. PyTorch. PyTorch Profiler Documentation, accessed 2024.
6. KV Sai Sundar, Lokeshwar Rao Bonta, Pallav Kumar Baruah, S Siva Sankara, et al. Evaluating training time of inception-v3 and resnet-50,101 models using tensorflow across cpu and gpu. In *2018 second international conference on electronics, communication and aerospace technology (ICECA)*, pages 1964–1968. IEEE, 2018.
7. Ultralytics. Ultralytics. <https://github.com/ultralytics/ultralytics>, accessed 2024.
8. Yong Shing Voon, Yunze Wu, Xinzhi Lin, and Kamran Siddique. Performance analysis of cpu, gpu and tpu for deep learning applications. *Professor Ka Lok Man, Xi’an Jiaotong-Liverpool University, China Professor Young B. Park, Dankook University, Korea Chairs of CICET 2021*, 16:12, 2021.

Note:

Query Mutation:

The use of **random transformations** in your DBMS fuzzing framework serves a critical purpose in testing database systems effectively. Here's a breakdown of why this approach is used:

**1. Simulating Real-World Query Variations**

Database queries in production systems are rarely written in a single "optimal" way. Developers often write logically equivalent queries with:

* Different operator orders (WHERE a > 5 AND b < 10 vs. WHERE b < 10 AND a > 5)
* Alternative syntax forms (BETWEEN vs. >=/<=, IN vs. OR)
* Varying projection orders (SELECT a, b vs. SELECT b, a)

**Randomization mimics this real-world diversity** to test how the DBMS handles equivalent but syntactically different queries.

**2. Uncovering Hidden Bugs**

Database engines often contain optimizers that rewrite queries internally. Subtle bugs can lurk in:

* Query optimization rules
* Operator precedence handling
* Join reordering logic

By **randomly altering query structure**, you force the DBMS to exercise different code paths, increasing the chance of triggering edge cases.

**3. Efficient Exploration of Possibilities**

Exhaustively testing all possible equivalent transformations would be computationally infeasible. For example:

* A query with 5 AND clauses has 5! = 120 possible orderings
* A SELECT with 4 columns has 4! = 24 projection orderings

**Random sampling** allows you to test a representative subset of these possibilities with limited resources.

**4. Equivalent Expression Transformation (EET) Philosophy**

Your framework specifically uses **EET-based fuzzing**, which relies on generating queries that are **logically equivalent** to the original. Randomness here ensures:

* The same semantic meaning (so discrepancies indicate bugs)
* Diverse syntactic representations (to stress-test the DBMS)

**Example Workflow**

For the seed query:

sql

Copy

SELECT name FROM users WHERE age > 25 AND salary < 100000

Random transformations might produce:

1. SELECT name FROM users WHERE salary < 100000 AND age > 25 (AND-swap)
2. SELECT name FROM users WHERE NOT (age <= 25 OR salary >= 100000) (De Morgan's Law)
3. SELECT name FROM users WHERE age > 25 AND salary BETWEEN 0 AND 99999 (BETWEEN conversion)

All are logically equivalent but test different DBMS internals.

**Why Not Systematic Transformations?**

While systematic testing (e.g., trying every possible transformation) seems ideal, it faces two key issues:

1. **Combinatorial Explosion**: Even moderately complex queries would generate thousands of variants.
2. **Resource Constraints**: Executing all variants would be time- and memory-intensive.

**Randomization + Parallel Execution** provides a practical balance between coverage and feasibility.

**Improving Randomization Quality**

To make randomness more effective:

1. **Weighted Transformations**: Prioritize high-value transformations (e.g., operator swaps over trivial changes)

python

Copy

self.transformations = [

(self.\_swap\_and\_clauses, 0.4), # 40% probability

(self.\_convert\_between, 0.3), # 30% probability

(self.\_reorder\_projections, 0.3) # 30% probability

]

1. **Feedback-Driven Fuzzing**: Track which transformations find bugs and prioritize them.
2. **Constraint Checks**: Ensure transformations don't break query validity.

**Key Takeaway**

Random transformations are **not arbitrary** – they’re a strategic tool to:

* Mimic real-world query diversity
* Efficiently explore equivalent query space
* Stress-test DBMS internals systematically

By combining randomness with equivalence guarantees (EET), your framework can uncover bugs that deterministic testing might miss.