Department of Computer Science and Engineering
School of Electrical and Computer Sciences
Indian Institute of Technology, Bhubaneswar

HIGH PERFORMANCE COMPUTER ARCHITECTURE

PROJECT PHASE 3

Analytical Modeling of LLM Computation and Communication on Real CPU/GPU Hardware

Team Members

Nali Bhavana - 24AI06013
Puvvula Nikhileswari - 24AI06017
Devesh Sharma - 24CS06002
Sapna Vishwakarma - 24CS06012

1 Motivation

Large Language Models (LLMs) have dramatically transformed the landscape of Natural Language Processing (NLP). These models have demonstrated human-like fluency and reasoning capabilities in tasks such as summarization, translation, question answering, generative writing, and many other tasks. However, the computational demands of these models are immense. High inference latency, large memory footprints, and significant communication overhead during training and inference phases pose challenges for deployment on real-world systems, especially those with limited resources like edge devices or shared cloud infrastructure.

With the exponential growth in model size and the increasing prevalence of transformer-based architectures, there is a critical need for infrastructure-aware optimizations. Organizations deploying LLMs face difficulties in estimating hardware requirements, costs, and energy consumption. Furthermore, the absence of a unified analytical model means developers often rely on empirical methods, which are time-consuming and hardware-specific. This project is motivated by the need to bridge this gap through a comprehensive profiling and modeling framework that helps predict the computational behavior of LLMs across diverse hardware configurations.

2 Problem Statement

Despite numerous optimizations such as quantization, pruning, model distillation, and speculative decoding, the underlying performance characteristics of LLMs remain opaque to system architects and developers.

Key challenges include:

Computation Complexity: LLMs involve deep transformer stacks with multi-head attention and large feedforward layers, leading to extensive floating-point operations.

Memory Constraints: Handling billions of parameters with high activation volumes often exceeds GPU memory capacity, causing Out-of-Memory (OOM) errors or excessive paging.

Communication Overhead: In distributed settings, data synchronization across GPUs or nodes via NCCL, NVLink, or InfiniBand introduces latency, especially when batch sizes or model sizes scale.

This project aims to systematically address the following questions:

- How does inference latency vary with model type, batch size, and workload?
- What are the dominant bottlenecks in LLM performance—compute, memory, or communication?
- Can we develop a predictive model to estimate latency, memory usage, and bandwidth utilization given a model configuration?
- Which design parameters most significantly impact runtime performance, and how can these insights guide hardware-aware optimizations?

3 Main Contributions

This work makes the following key contributions:

Profiling Framework

Developed a modular PyTorch-based framework to profile LLMs using `torch.profiler`, CUDA event timing, and NVML memory tracking. The profiler captures latency, peak memory, FLOPs, and memory bandwidth.

Dataset Creation

Compiled an extensive dataset by profiling four transformer-based models (OPT-350M, BLOOM-560M, T5-Small, BART-Base) across various NLP tasks (summarization, translation, QA, generation) and batch sizes (1, 8, 32).

Analytical Model

Trained a Random Forest-based regressor to predict performance metrics (latency, memory usage, and bandwidth) from inputs like batch size, model parameters, architecture type, and workload type. Features were log-transformed and encoded to improve prediction accuracy.

Visualization Suite

Built visualization tools to analyze predicted vs actual latency, residuals, feature importances, and batch-scaling trends, helping interpret model behavior and identify optimization targets.

Empirical Insights

Observed that decoder-only models are more latency-efficient in generation and QA tasks, while encoder-decoder models offer better accuracy in summarization and translation at the cost of increased CPU utilization and latency.

Executable Pipeline

A fully automated Python pipeline was implemented to execute the profiling, train the model, generate evaluation metrics, and visualize results. A sample configuration demonstrates prediction capability for new model setups.

4 Experimental Setup

Hardware Platform

- NVIDIA Tesla T4 GPU (via Google Colab)

Models Evaluated

- **Decoder-only:** `opt-350m`, `bloom-560m`
- **Encoder-decoder:** `t5-small`, `bart-base`

Workloads Tested

- Summarization
- Translation
- Question Answering
- Generation

Batch Sizes

- 1, 8, 32, 128

Profiling Tools

- `torch.profiler` for CUDA kernel activity and memory tracking
- `pynvml` for GPU utilization and memory info
- `scikit-learn` for model training and evaluation
- `matplotlib`, `seaborn` for result visualization

5 Procedure

The procedure adopted for profiling and analytical modeling of LLM performance involves the following steps:

1. Model Initialization

Each of the selected transformer-based models is loaded along with its corresponding tokenizer using the HuggingFace Transformers library. The models are then transferred to the GPU (NVIDIA Tesla T4) for accelerated computation. The models include:

- **Decoder-only models:** `opt-350m`, `bloom-560m`
- **Encoder-decoder models:** `t5-small`, `bart-base`

2. Input Tokenization

For each workload (Summarization, Translation, Question Answering, Generation), synthetic input text is tokenized using the corresponding tokenizer. The tokenized input is padded or truncated to a maximum length of 512 tokens to maintain consistency across workloads and batch sizes (1, 8, 32, 128).

3. Profiling Execution

The inference process is profiled using the following tools:

- `torch.profiler` to record CUDA kernel execution, latency, memory operations, and FLOPs.
- `pynvml` for GPU-level metrics such as utilization and peak memory.

Each profiling run involves explicit CUDA memory synchronization before and after the forward pass to ensure accurate measurement of:

- **Latency:** Measured using CUDA events and wall-clock time.
- **Peak Memory Usage:** Tracked using NVML queries.
- **Memory Bandwidth:** Estimated from profiler data linked to DRAM reads/writes.

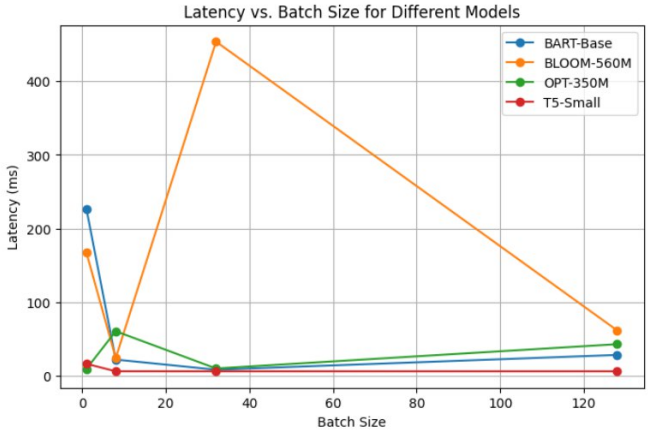

Figure 1: Latency vs Batch Size for different Models

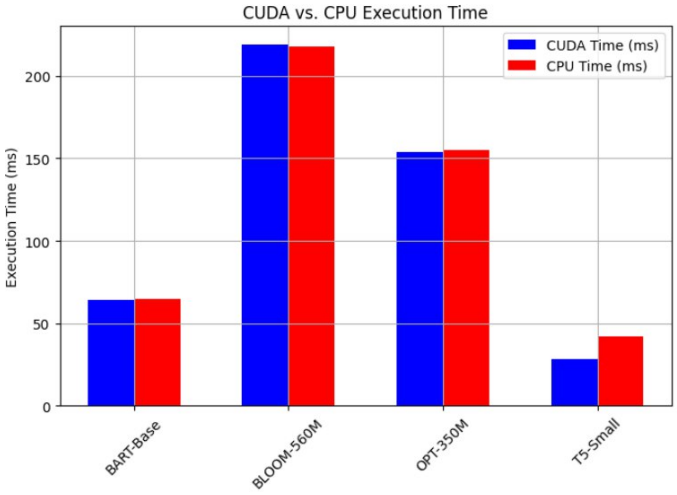

Figure 2: CUDA VS CPU Execution Time Across Models

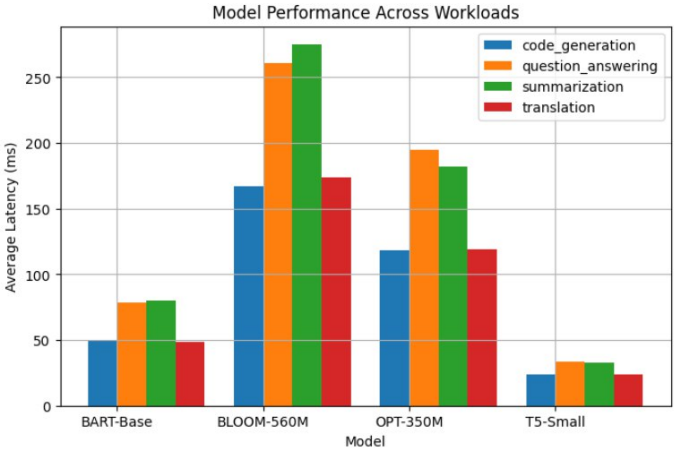

Figure 3: Latency Analysis Across Models

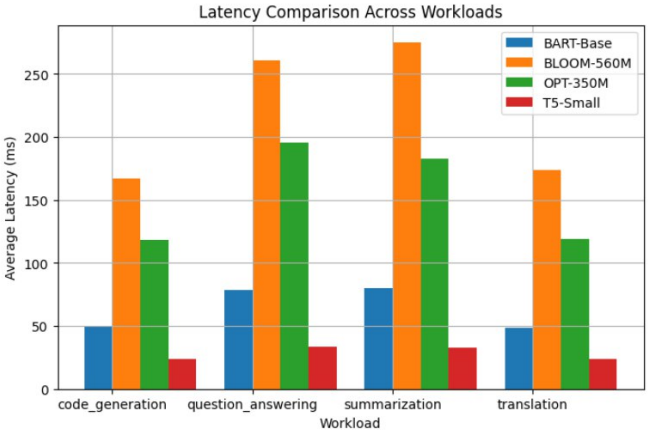

Figure 4: Latency Analysis Across Workloads

4. Data Collection

The raw profiling data (latency, memory, FLOPs, bandwidth) is aggregated and exported to CSV format. Each row in the dataset includes:

- Model type, architecture, and parameter count
- Batch size and workload type
- Observed latency (in seconds)
- Peak GPU memory usage (in MB)
- Estimated memory bandwidth (in arbitrary units)

5. Analytical Modeling

The dataset is used to train a regression model (Random Forest) using `scikit-learn`. Key steps include:

- Log transformation and encoding of features to handle scale and categorical attributes
- Training and cross-validation using metrics such as MAE, RMSE, and R^2
- Model evaluation and comparison of predicted vs actual values

6. Visualization

Using `matplotlib` and `seaborn`, a visualization suite is built to:

- Compare predicted and actual latency
- Analyze residuals and feature importances
- Visualize trends across batch sizes and workloads

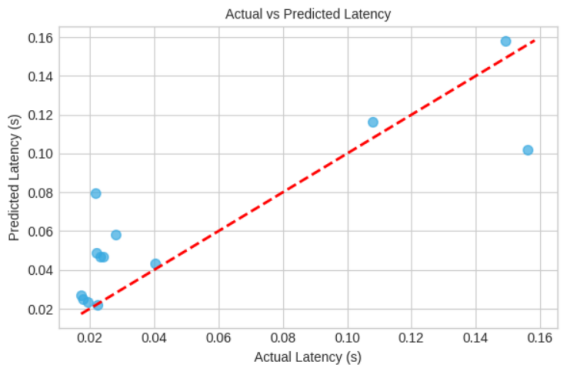

Figure 5: Actual vs Predicted Latency

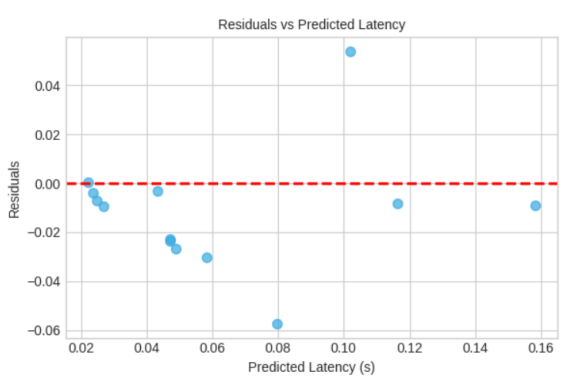

Figure 6: Residuals vs Predicted Latency

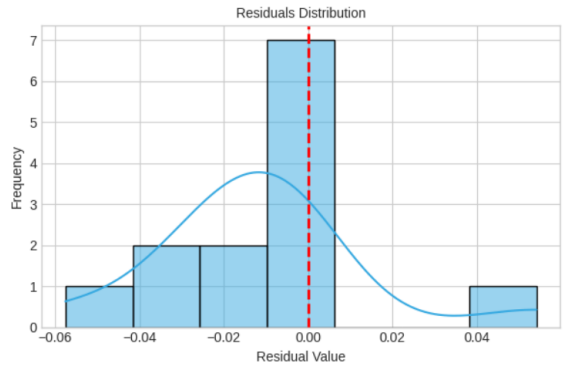

Figure 7: Residuals Distribution

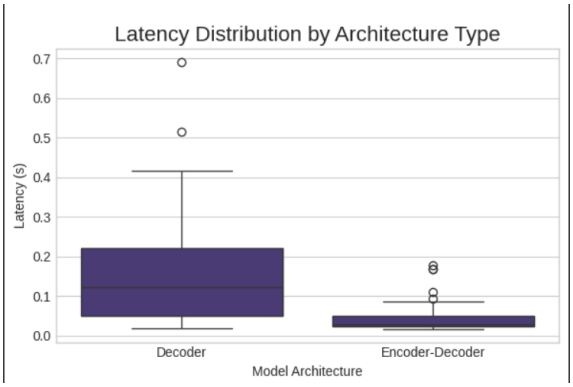

Figure 8: Latency Distribution By Architecture Type

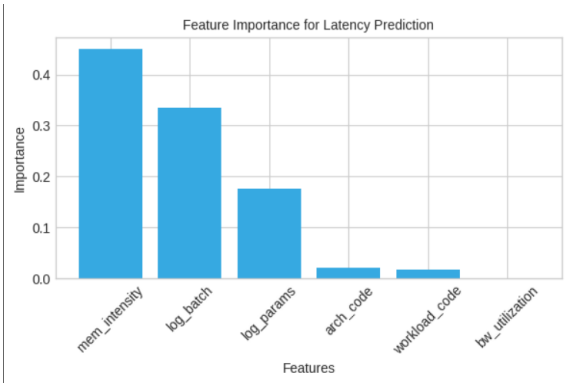

Figure 9: Important features

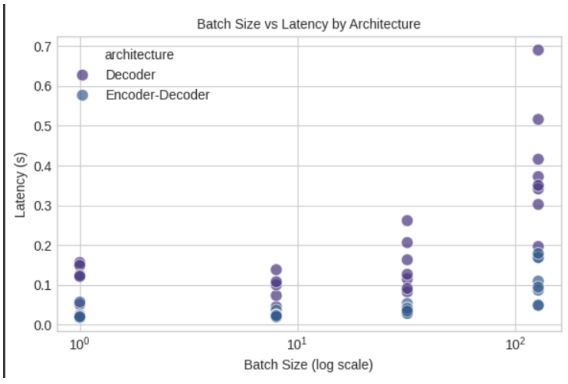

Figure 10: Batch Size vs Latency by Architecture

7. Automation

The entire profiling-to-modeling pipeline is implemented in Python, making it modular and reproducible. A sample configuration allows performance prediction for new model and workload combinations, enabling rapid evaluation and optimization insights.

6 Evaluation Criteria

The effectiveness of the profiling and analytical modeling framework was assessed based on:

Latency Measurement

Wall-clock time for forward pass using CUDA events, converted to seconds.

Peak Memory Usage

Maximum memory allocated on GPU during execution, measured via NVML.

Memory Bandwidth Estimation

Aggregated from profiling FLOPs linked to DRAM reads and writes.

Model Parameters

Total number of learnable parameters used as a proxy for computational complexity.

Model Accuracy (Prediction)

Evaluated using metrics like Mean Absolute Error (MAE), Root Mean Square Error (RMSE), and R-squared (R^2) to quantify how well the model predicts latency and other metrics.

Table 1: Prediction Accuracy Metrics

Metric	MAE	RMSE	R^2
Latency (s)	0.0197	0.0267	0.7080
Peak Memory (GB)	0.2047 GB	0.4848 GB	0.9374
Bandwidth (AU)	0.0	0.0	1.0

7 Results and Discussion

7.1 Latency Trends

Latency increases with batch size, with a steeper growth observed in encoder-decoder models due to additional decoder-side attention computations. In contrast, decoder-only models demonstrate better scaling characteristics with increasing batch size, making them more suitable for low-latency applications.

7.2 CPU vs CUDA Time

Encoder-decoder models tend to offload a greater portion of computation to the CPU, resulting in reduced GPU efficiency. Conversely, models such as OPT and BLOOM leverage CUDA acceleration more effectively, exhibiting fewer CPU bottlenecks and improved end-to-end inference times.

7.3 Workload Differences

- **Summarization/Translation:** T5 and BART yield better quality but are slower.
- **QA/Generation:** OPT and BLOOM offer faster inference and lower memory use, ideal for real-time tasks.

7.4 Feature Importance and Predictions

Analysis reveals that the logarithm of batch size and the number of model parameters are the most dominant features in predicting latency. Furthermore, the architecture type—whether decoder or encoder-decoder—has a substantial influence on model performance.

Predictions from the Random Forest regression model align closely with the actual latency values, as evidenced by the tight clustering around the ideal line in the *Actual vs. Predicted* plot (Figure 5).

7.5 Visualization Summary

The scatter plots and residual histograms validate the accuracy and reliability of the regression model. Boxplots illustrate the distribution of latency across different architecture types. Finally, feature importance plots effectively identify the critical factors driving inference latency (Figures 6,7 and Figure 9).

8 Conclusion and Future Work

This work presents a scalable pipeline for profiling and modeling LLM inference on GPUs. By capturing execution traces and training predictive models, we can estimate resource needs and spot inefficiencies before deployment, enabling better planning for latency-sensitive tasks.

Future Work

Key directions include:

- **Larger Models:** Extend profiling to models like LLaMA, Falcon, and GPT-NeoX.
- **Energy Metrics:** Add power tracking using `nvidia-smi dmon` or external meters.
- **New GPUs:** Validate predictions on A100, H100, and RTX 4090.
- **Optimizations:** Apply quantization, distillation, and dynamic batching.
- **Simulation:** Integrate with simulators like Vidur for inference planning.

This study supports efficient, sustainable LLM deployment and aligns with Green AI goals.

References

1 Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav Gulavani, Ramachandran Ramjee, Alexey Tumanov. "Vidur: A Large-Scale Simulation Framework For LLM Inference."