VASAVI COLLEGE OF ENGINEERING
IBRAHIMBAGH, HYDERABAD

IMPLEMENTATION OF LARGE LANGUAGE MODEL ON DIVERSE HARDWARE PLATFORMS

A PROJECT REPORT

Submitted By:

1602-22-735-099 Yennam Sai Tharun Reddy

1602-22-735-118 P.Supriya Sree

1602-22-735-103 Uduthanaboina Sathwik

Submitted To:

Department of Electronics Communication Engineering

Vasavi College of Engineering

Imbrahimbagh, Hyderabad

500031

JUNE, 2024

DECLARATION

We, the undersigned , declare that the project report **Implementation of Large Language Model on Diverse Hardware Platforms** submitted for partial fulfillment of the requirements for the award of the degree of Bachelor of Engineering of Vasavi College of Engineering(A), Telangana, is a work done by me under the supervision of Dr.A.Srilakshmi, Mr.V.Krishina Mohan . I here by affirm that this submission is a true representation of my own ideas and words. In instances where I have incorporated the ideas or words of others, I have diligently and accurately cited and referenced the sources. Furthermore, I confirm my adherence to the principles of academic honesty and integrity, ensuring that no data, idea, fact, or source has been misrepresented or fabricated in my submission. I fully comprehend that any deviation from aforementioned standards may result in disciplinary action by the institute and/or the University, and may also prompt legal action from individuals whose work has not been appropriately cited or for which proper permission has not been obtained. Additionally , I declare that this report has not been previously used as the basis for the award of any degree, diploma, or similar title from other University .....

Name : Y.Sai Tharun Reddy
Signature:.....
Name : P.Supriya Sree
Signature:.....
Name : U.Sathwik
Signature:.....

DEPARTMENT OF ELECTRONICS COMMUNICATION ENGINEERING

Vasavi College of Engineering
Ibrahimbagh,Hyderabad
500031

CERTIFICATE

This is to certify that the report entitled **IMPLEMENTATION OF LARGE LANGUAGE MODEL ON DIVERSE HARDWARE PLATFORMS** Submitted by **Y.Sai Tharun Reddy,P.Supriya Sree,U.Sathwik** to the Vasavi College of Engineering in partial fulfillment of the requirements for the award of the Degree of Bachelor of Technology in Electronics Communication Engineering is a bonafide record of the project work carried out by him/her under my/our guidance and supervision. This report in any form has not been submitted to any other University or Institute for any purpose.

Project Coordinator:
Name : Dr.A.Srilakshmi
Signature:.....

Project Coordinator:
Name : Mr.V.Krishna Mohan
Signature:.....

Head of Department
Name : Dr.E Sreenivasa Rao
Signature:.....

ACKNOWLEDGEMENT

I wish to record my indebtedness and thankfulness to all who helped me to prepare this Project titled **LARGE LANGUAGE MODEL ON DIVERSE HARDWARE PLATFORMS** and present it satisfactorily

I am especially thankful for my guides Dr .A.Srilakshmi and Mr. V.Krishna Mohan in the Department of Electronics Communication of Engineering for giving me valuable suggestions and critical inputs in the preperation of this report. I am also thankful to Dr.E Sreenivasa Rao, Head of Department of Electronics Communication of Engineering for encouragement

Thank you all for your assistance and support.

Y. sai Tharun Reddy
P. Supriya Sree
U. Sathwik
B.E(Electronics Communication of Engineering)
Department of Electronics Communication of Engineering
Vasavi College of Engineering

Abstract

Large language model (LLM) deployment across heterogeneous hardware platforms is a major obstacle to modern artificial intelligence (AI) research and practice. Even while LLMs have proven to be unmatched in their ability to comprehend and generate natural language, there is still significant worry regarding their effective deployment and use on a variety of hardware architectures, such as CPUs, GPUs, and FPGAs. It is imperative to optimize LLMs for various hardware configurations in order to guarantee scalability, performance, affordability, and energy efficiency. In addition, it is imperative to tackle the challenges associated with implementing LLMs on heterogeneous hardware platforms in order to facilitate edge computing, real-time applications, and universal access to cutting-edge AI technologies in a variety of settings and user bases. Thus, the core of our research problem is to comprehend the nuances of LLM deployment on various hardware architectures, with the goal of realizing the full potential of these transformational models while tackling the pragmatic issues related to their use and implementation

TABLE OF CONTENTS

1. INTRODUCTION

According to this paper, large language models (LLMs), including the well-known Transformer-based models, have revolutionized a number of fields by demonstrating exceptional performance on tasks involving the generation and processing of natural language. However, because these models have large compute and memory overheads,efficiently deploying them is difficult.

1.1. Efficiency Enhancement:

- Large computational resources are needed for inference with LLMs. FPGAs have the potential to speed up LLM inference by taking advantage of their parallelism and scalability.
- Using FPGA-specific resources like DSP48 blocks and heterogeneous memory, FlightLLM, for example, offers effective LLM inference on FPGAs.
- The objective is to outperform commercial GPUs in terms of cost and energy efficiency.

1.2. Motivation

Our motivation to work on this Testing large language models (LLMs) on various hardware platforms such as CPUs, GPUs, and FPGAs since we need to maximize energy efficiency, performance, and cost. Through the assessment of these heterogeneous platforms, we are able to determine the optimal configurations for different applications, guaranteeing that LLMs function at maximum efficiency. By using energy-efficient solutions, this strategy not only encourages sustainable AI practices but also increases accessibility and inclusivity for cutting-edge AI technologies. Furthermore, our research encourages innovation in software and hardware, setting us up for future developments and guaranteeing the continued stability, scalability, and versatility of our AI deployments.

1.3. Objectives

The goal of this research is to maximize the use of large language models (LLMs) across a range of hardware platforms, including as FPGAs, GPUs, and CPUs. Among our goals are

- **Performance Optimization**: Assess LLM's performance on various hardware platforms to find the best setups for applications like text generation and classification.
- **Cost-Efficiency**: Examine cost-effectiveness by taking maintenance costs, energy usage, and hardware procurement into account.

- **Energy Efficiency**: Look into ways to minimize the impact on the environment and operating expenses by consuming less energy throughout LLM deployment.
- **Real-Time Applications**: Consider putting LLMs on edge devices for chatbots and voice assistants, among other real-time applications.
- **Inclusivity**: Make sure that advanced AI technologies are accessible to users with a range of abilities by ensuring accessibility across various hardware.

2. LITERATURE SURVEY

This section of our project report presents the research done for the project

Large Language Models (LLMs) such as GPT-3 have become essential for processing and producing text that is human-like in the field of artificial intelligence. Because of the high computing demands of these models, there is a lot of research being done on the implementation of LLMs on various hardware platforms. The results of recent research on the implementation of LLMs on several hardware platforms, such as CPUs, GPUs, and FPGAs, are summarized in this literature review.

Although CPUs are the most widely available hardware, they do not have the parallel processing power required for large-scale manufacturing (LLM) processes. Despite this, because of their widespread usage in the current infrastructure, they are nevertheless employed for smaller-scale LLM applications.

GPUs are now the go-to option for LLM training and inference due to their unmatched throughput and parallelism. Their design is ideal for LLMs since it is compatible with matrix and tensor operations. However, the high power consumption of GPUs raises several issues, particularly when training longer and larger models

Because they are customizable, FPGAs provide a middle ground by enabling customized optimizations for particular LLM workloads. They may offer more energy efficiency than GPUs, which makes them a desirable choice for long-term, extensive LLM deployments. However, the extended development cycles and intricate programming of FPGAs may prevent their widespread use.

Custom- designed infrastructures, similar ASICs, have also been the subject of recent exploration since they can give LLMs with the loftiest degree of performance optimization. These technical tackle options can lower quiescence and increase energy frugality dramatically. still, the operation of custom- designed infrastructures is confined to high-budget exploration and marketable operations due to their high cost and lack of flexibility.

The literature also emphasizes how pivotal performance measures for assessing tackle platforms for LLMs are, including outturn, quiescence, energy frugality, and cost effectiveness. The stylish tackle is chosen grounded on these characteristics for particular LLM operations.

To sum up, the field of implementing LLMs on various hardware platforms is dynamic and demands a balance between cost, energy usage, and performance. The creation of specialized hardware accelerators will be essential to facilitating LLMs' wider acceptance and incorporation into practical applications as they continue to expand in size and complexity12. This survey sheds insight on the trade-offs and factors to take into account while deploying these models on various platforms by offering a succinct summary of the state of hardware accelerators for LLMs.

3. DESIGN AND METHODOLOGY

- **HUGGING FACE MODEL**
- **USING JUPITER SOFTWARE APPROACH**
- **CONVERSION TO ONNX MODEL**
- **IMPLEMENTATION ON DIFFERENT HARDWARE LIKE CPU,GPU,FPGA**

4. Large Language Model

In the area of artificial intelligence (AI), large language models (LLMs) are a revolutionary development, especially in natural language processing (NLP). These advanced models are made to understand, produce, and work with human language with astounding coherence and accuracy. Large-scale legal databases and significant processing capacity have made LLMs indispensable for a wide range of uses, such as content production, automated customer support, legal document analysis, and scientific research. The transition from conventional rule-based systems and statistical techniques to cutting-edge deep learning methodologies has defined the evolution of LLMs. The creation of transformer architectures, as demonstrated by models like OpenAI's GPT-4, Google's BERT, and T5, is one of the major advances. Because of these structures, LLMs are able to comprehend the subtleties and context of language, which enables them to carry out a variety of intricate linguistic tasks. Large volumes of textual data from many sources are processed throughout the training phase of these models, which aids in their deepening comprehension of language, including syntax, semantics, and cultural quirks. Because of this, LLMs are excellent at tasks like sentiment analysis, question answering, summarization, text production, and translation

4.1. Transformer architecture:

The basis for contemporary large language models (LLMs) like Google's BERT, OpenAI's GPT-4, and T5 is the transformer architecture. The transformer model, which was first presented in the 2017 paper "Attention is All You Need" by Vaswani et al., has completely changed natural language processing (NLP) by facilitating effective training and exceptional performance on a range of language tasks. Important Elements of the Transformer: Self-Attention Mechanism When encoding a specific word, the model can consider the relative relevance of various words in a phrase thanks to self-attention. The shortcomings of earlier recurrent and convolutional neural network-based models are addressed by this process, which aids in capturing relationships between words regardless of their distance from one another in the text. Reduced Dot-Product After using attention to calculate attention scores between word pairs, attention weights are obtained by using a softmax function. The weighted total of the input values is then calculated using these weights, emphasizing pertinent terms.

4.2. Multiple Head attention:

The transformer model uses multi-head attention, which is the simultaneous use of several self-attention mechanisms. By focusing on many phrase components at once, this enables the model to capture all kinds of links and dependencies in the data. The model is able to gather richer data by having each head run independently and then concatenate and change their outputs linearly.

4.3. Positional encoding:

Transformers do not analyze data sequentially, in contrast to recurrent models. Positional encoding, which adds information about each word's position to the input embeddings, helps to preserve the sentence's word order. Sinusoidal functions are used to produce this encoding, guaranteeing that the model can discern between various sequence places.

4.4. Feed forward network:

A feed-forward neural network is applied independently to each point in each layer of the transformer. These networks allow the model to catch intricate patterns in the data by combining two linear transformations with a ReLU activation in between.

4.5. Layer Normalization and Residual Connections:

The training process is accelerated and stabilized by applying layer normalization. By enabling gradients to pass straight through the network, residual connections aid in reducing the vanishing gradient issue and enhance the learning of deeper models. Structure of transformer architecture: Although several LLMs only use the encoder (BERT) or the decoder (GPT) for particular tasks, the transformer design is made up of an encoder-decoder structure. **Encoder**: The encoder is made up of several identical layers, one feed-forward network and one multi-head self-attention mechanism per layer. Rich contextual representations are produced by the encoder through processing of the input text. **Decoder:** Comprising several identical layers as well, the decoder has an extra multi-head attention mechanism over the encoder's output that enables it to concentrate on pertinent segments of the input sequence while producing the output.

4.6. Advantages of transformer architecture:

Parallelization: Transformers, as opposed to recurrent models, enable the processing of sequence data in parallel, greatly accelerating training and inference times. **Managing Long-Term Reliances:** Transformers are able to grasp long-range dependencies in text, something that regular RNNs find difficult. This is because of self-attention processes. **Scalability:** Training big language models requires an architecture that works well with big datasets and model sizes. A helpful tool for applications like chatbots, machine translation, text summarization, and many other language-related tasks, LLMs can give more accurate and contextually relevant responses by utilizing the transformer architecture. It is anticipated that as research proceeds, transformer-based models will undergo additional improvements and modifications that will push the limits of what is achievable in natural language processing.

5. HUGGING FACE

Hugging Face supports the creation and application of Large Language Models (LLMs) in a number of ways.

Model Repository: Offers a location to save and find LLMs, making pretrained models easily accessible for applications such as summarization, translation, and text generation. Dataset sharing allows researchers to train and fine-tune LLMs on a variety of data sources by providing a broad range of datasets pertinent to language problems. Reduces the computational load on researchers and developers by providing compute solutions, including GPU support, for training and inference activities.

Transformers Library: Hugging Face's seminal open-source project, the Transformers library streamlines LLM creation by offering pre-trained models and cutting-edge LLM designs for PyTorch, TensorFlow, and JAX. Hugging Face offers fast tokenizers that improve the effectiveness of LLM training and inference, particularly for large-scale language datasets.

Open Source Contributions: Consistently advances LLM research by providing state-of-the-art structures and methodologies for natural language processing jobs through open-source initiatives such as Transformers.

Conversion of hugging face to onnx model:

When incorporating a Hugging Face model into a project, interoperability between various deep learning frameworks is facilitated by converting it to the Open Neural Network Exchange (ONNX) standard. A thorough tutorial on transforming a Hugging Face model to ONNX may be found below:

preparaing the model:

Make sure the model for the Hugging Face is loaded and prepared for conversion. The onnx and onnxruntime libraries for Python must be installed in order to complete the conversion.

converting the model:

Create the Hugging Face model object, then add weight to it. To convert the model to ONNX format, use the torch.onnx.export() function (provided you have a PyTorch backend). Along with any other pertinent characteristics, provide the forms of the input and output.

import torch
import onnx
from transformers import AutoTokenizer, AutoModelForSequenceClassification
modelname = ”nlptown/bert-base-multilingual-uncased-sentiment”
tokenizer = AutoTokenizer.frompretrained(modelname)
model = AutoModelForSequenceClassification.frompretrained(modelname)
inputsentence = ”I love this product!”
inputids = tokenizer(inputsentence, returntensors=”pt”)[”inputids”]
model.to(”cpu”) Ensure the model is on CPU for export
onnxpath = ”yourmodel.onnx” Specify the output path for the ONNX file
torch.onnx.export(model,(inputids), onnxpath, opsetversion=11)
print(”success:”,onnxpath))

Validation:

Use the ONNX model checker tool (onnx.checker.checkmodel()) to validate the exported ONNX model. Verify that the model structure is free of mistakes and inconsistencies.

import onnx

Load the ONNX model
onnxmodel = onnx.load(”huggingfacemodel.onnx”)
Check the ONNX model for errors
onnx.checker.checkmodel(onnxmodel)

6. ONNX MODEL(Open Neural Network Exchange)

ONNX (Open Neural Network Exchange) is an intermediary machine learning framework designed to facilitate seamless conversion between different machine learning frameworks.

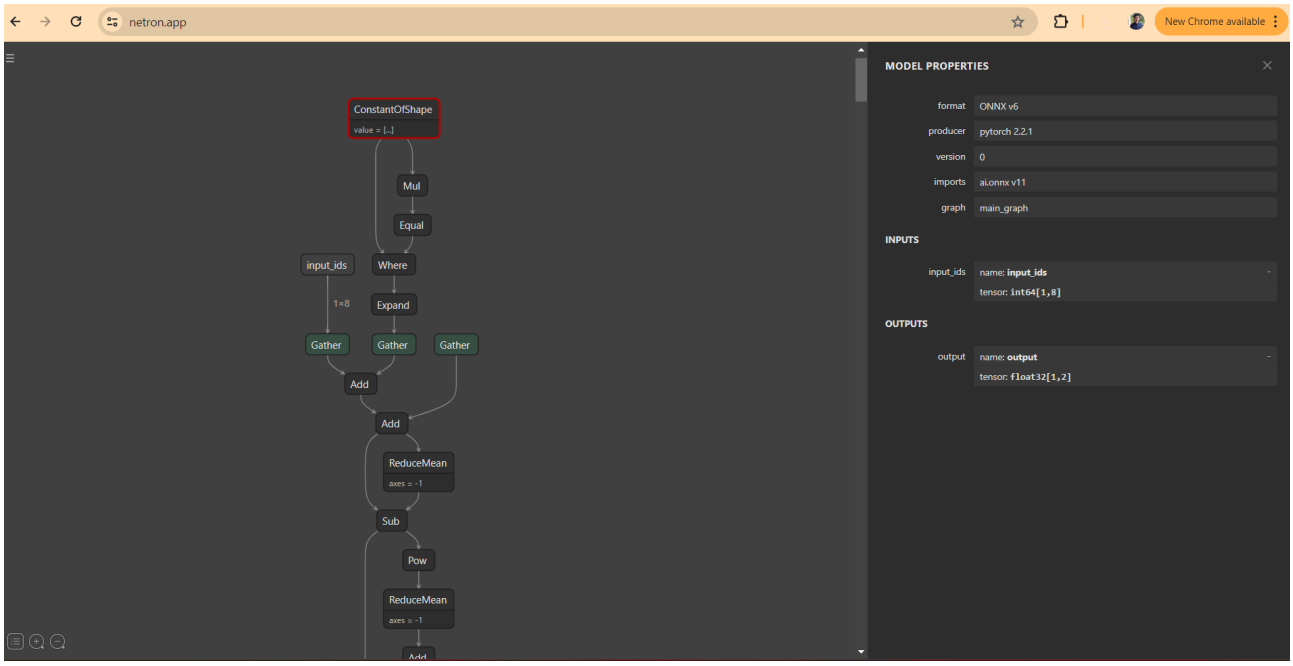
1. Purpose and Functionality: o Imagine you're working with a model in one framework (e.g., TensorFlow), but you need to deploy it in a different framework (e.g., TensorRT or TFLite). o ONNX acts as a bridge, allowing you to convert your model from one framework to another. o It supports interoperability by providing a common format for representing models. Well known systems and apparatuses congruous with ONNX. ONNX's capacity to effectively plug into systems and devices as of now utilized in a wide assortment of applications highlights its esteem.

This compatibility permits AI engineers to use the qualities of distinctive stages whereas keeping their models convenient and effective. Pytorch. A broadly utilized open-source machine learning library from Facebook. Pytorch's notoriety infers from its ease of utilize and energetic computational charts. The community favors PyTorch for investigate and improvement due to its adaptability and instinctive plan. TensorFlow. TensorFlow could be a total system created by Google. TensorFlow gives high-level and low-level APIs for building and sending machine learning models. Microsoft Cognitive Toolkit (CNTK). A profound learning system from Microsoft. CNTK is known for its viability in preparing convolutional neural systems and is especially well suited for discourse and picture acknowledgment errands. Apache MXNet. A adaptable and productive open-source profound learning system from Amazon. MXNet conveys profound neural systems over numerous stages, from cloud foundation to portable gadgets. Scikit Learn. A well known library for conventional machine learning calculations. Whereas not straightforwardly congruous, scikit-learn models can be changed over utilizing sklearn-onnx. Camera. Keras, a high-level neural arrange API, runs on beat of TensorFlow, CNTK, and Theano. The reason is to empower fast testing of Apple Center ML. Models can be changed over from other systems in ONNX arrange to Center ML and coordinates into iOS applications. Run ONNX. A effective cross-platform assessment apparatus. Optimizes demonstrate induction on equipment and is basic for arrangement. NVIDIA TensorRT. SDK for powerful deep learning deduction. TensorRT incorporates an ONNX parser and is utilized for optimized induction on NVIDIA GPUs. ONNX.js. JavaScript library for running ONNX models within the browser and on Node.js. Empowers the utilize of ONNX models in web-based ML applications

2. How It Works: o Train your model in a popular framework like PyTorch, TensorFlow, or MXNet. o Convert the trained model into the ONNX format. o Now you can consume this ONNX model in a different framework (e.g., ML.NET or other machine learning libraries). 3. Benefits: o Framework Agnostic: ONNX promotes compatibility among various deep learning frameworks. o Transferability: You can move models seamlessly between different frameworks without retraining. o Standardization: ONNX provides a universal representation of computational graphs.

6.1. NETRON

Netron is an open-source viewer for neural network, deep learning, and machine learning models. It is designed to help users visualize, explore, and understand the structure and components of their models. Netron supports a wide range of model formats, making it a versatile tool for developers and researchers in the field of machine learning. Key Features Model Visualization: Graphical Interface: Netron provides a visual representation of neural network models, displaying the architecture as a graph with nodes representing different layers and connections showing the data flow. Detailed Layer Information: Users can click on each node to see detailed information about that layer, including input/output shapes, parameters, and attributes. We have used this for Analysing ONNX Model



7. DIFFERENT HARDWARE

CPU's: Central Processing Units (CPUs) are the core components of computer systems, carrying out calculations and instructions. As GPUs and TPUs do in the world of large language models (LLMs), CPUs are essential. Parallel computing is where GPUs and TPUs shine, while CPUs are more flexible, dependable, and widely available. CPUs are used in LLM deployment for operations where parallelization is not as important, such as data preprocessing, model parameter management, and inference workloads. They are crucial in situations where resource limits, latency, and power consumption are given top priority, notably in edge computing environments. Although CPUs are not as powerful as GPUs and TPUs, they nevertheless offer the flexibility and compatibility needed to implement LLMs on a variety of hardware platforms.

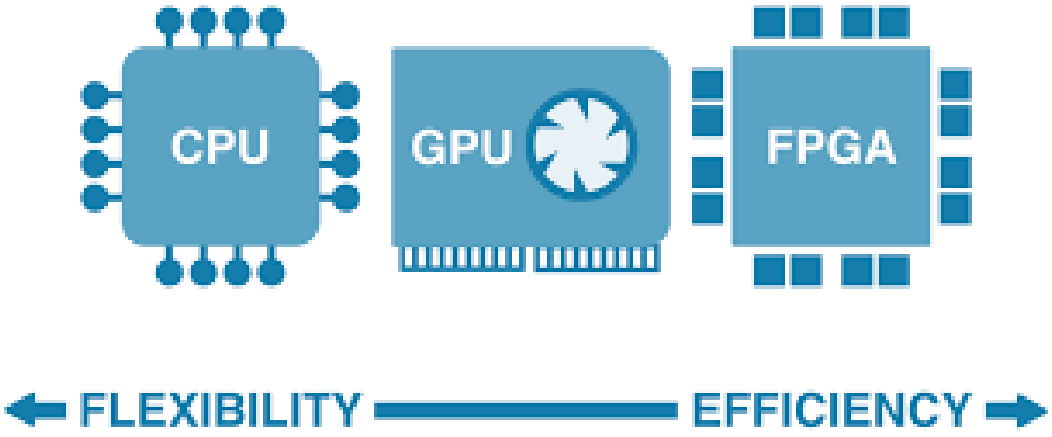
GPU's: Large language models (LLMs) require the use of Graphics Processing Units (GPUs), which are specialized hardware components made for parallel processing jobs. Because of their massively parallel architecture, GPUs are excellent for speeding up computationally demanding processes in LLM deployment, like model training and inference. Through the simultaneous distribution of calculations over thousands of cores, GPUs drastically shorten LLM training durations. Faster experimentation and iterations are made possible by this parallelism, which spurs creativity in model building and optimization. GPUs also play a key role in the deployment of LLMs for real-time applications like natural language production and interpretation jobs, which call for the quick processing of big datasets. Due to their tremendous processing throughput and the latest developments in GPU technology, they are the go-to option for optimizing performance in LLM deployment settings and speeding up AI applications.

FPGA: Large language models (LLMs) can be deployed with special advantages because to reconfigurable technology called field programmable gate arrays (FPGAs). In contrast to CPUs and GPUs, which have fixed architectures, FPGAs are programmable and customizable to perform particular jobs well. FPGAs provide advantages including low latency, fast throughput, and energy efficiency in LLM deployment. They are ideal for expediting complex calculations involved in model inference and data preparation chores because of their parallel processing capabilities and capacity to construct specialized hardware accelerators. FPGAs are very useful in edge computing settings where space and power constraints are important factors to take into account. LLMs can achieve real-time performance and responsiveness by shifting compute-intensive operations to FPGAs. This makes them appropriate for low latency processing applications like speech recognition and natural language comprehension.

COMPARISION BETWEEN CPU,GPU and FPGA:

CPUs, **GPUs**, and **FPGAs** in the context of working with **Large Language Models (LLMs)**.

Aspect	CPU	GPU	FPGA
Parallelism	Limited parallelism. Suitable for general-purpose tasks.	High parallelism with thousands of cores. Ideal for matrix operations.	Customizable parallelism. Can be tailored to specific LLM workloads.
Memory Hierarchy	Hierarchical memory (cache, RAM).	High memory bandwidth. GDDR for GPU.	Customizable memory hierarchy. Can optimize for LLM data access patterns.
Latency	Higher latency due to sequential execution.	Lower latency due to parallelism.	Low latency with custom data paths.
Throughput	Moderate throughput for serial tasks.	High throughput for parallel tasks.	Configurable throughput based on design.
Energy Efficiency	Efficient for serial workloads.	Less energy-efficient due to high power consumption.	Energy-efficient when optimized for specific tasks.
Customization	Not customizable beyond software optimizations.	Limited customization (e.g., CUDA kernels).	Highly customizable using hardware description languages.
Ease of Programming	High-level languages (C/C++, Python).	CUDA, OpenCL, or specialized libraries.	Requires hardware description languages (VHDL, Verilog).
Deployment Cost	Affordable, widely available.	Expensive upfront cost.	Moderate upfront cost, but can be cost-effective for specific workloads.
Scalability	Limited scalability for parallel tasks.	Scalable for parallel tasks across multiple GPUs.	Scalable with custom designs for large-scale deployment.



8. IMPLEMENTATION

Loading Hugging Face Model and implementation code:

```python
from transformers import TFAutoModelForSequenceClassification, AutoTokenizer
import tensorflow as tf
import time

# Load the model and tokenizer
model_name = "nlptown/bert-base-multilingual-uncased-sentiment"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = TFAutoModelForSequenceClassification.from_pretrained(model_name)

# Example text
text = "What is capital of India?"
length=len(text)
print(f"Length of the string: {length}")
# Tokenize input text
inputs = tokenizer(text, return_tensors="tf", padding=True, truncation=True)

# Measure inference time for a single example
start_time = time.time()
_ = model(**inputs)
end_time = time.time()

latency_ms = (end_time - start_time) * 1000
print(f"Latency: {latency_ms:.2f} ms")

# Measure inference time for multiple examples
num_examples = 100
start_time = time.time()
for _ in range(num_examples):
    _ = model(**inputs)
end_time = time.time()

total_inference_time = end_time - start_time
throughput = num_examples / total_inference_time
print(f"Throughput: {throughput:.2f} examples/second")
```

converting into Onnx format and implementation code:

```python
import torch
import onnx
from transformers import AutoTokenizer, AutoModelForSequenceClassification

model_name = "nlptown/bert-base-multilingual-uncased-sentiment"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSequenceClassification.from_pretrained(model_name)
input_sentence = "I love this product!"
input_ids = tokenizer(input_sentence, return_tensors="pt")["input_ids"]

model.to("cpu")  # Ensure the model is on CPU for export
onnx_path = "yourmodel.onnx"  # Specify the output path for the ONNX file
torch.onnx.export(model,(input_ids), onnx_path, opset_version=11)
print("success:",onnx_path)
```

```python
import onnxruntime
import numpy as np
from transformers import BertTokenizer
 # Load the ONNX model
onnx_model_path = "./yourmodel.onnx"
ort_session = onnxruntime.InferenceSession(onnx_model_path)
 # Tokenize inp
tokenizer = BertTokenizer.from_pretrained('nlptown/bert-base-multilingual-uncased-sentiment')
input_sentence = "What capital of India?"
length=len(input_sentence)
print(f"Length of the string: {length}")
input_ids_1= tokenizer.encode(input_sentence, return_tensors="np", padding=True,
truncation=True)
input_ids_1 = np.array(input_ids_1, dtype=np.int64)
 # Run the model
ort_inputs = {"input.1": input_ids_1}
ort_outs = ort_session.run(None, ort_inputs)
  # Get the prediction
predicted_class = np.argmax(ort_outs[0])
print("Predicted class:", predicted_class)
import time
start_time = time.time()
output = ort_session.run(None, {ort_session.get_inputs()[0].name:  input_ids_1})
end_time = time.time()
# Calculate latency
latency_ms = (end_time-start_time) *1000
print(f"Latency: {latency_ms:.2f} ms")
```

```python
# Generate input data (you need to replace this with your actual input data)
input_shape = (1, 7)  # Assuming input shape is (batch_size, sequence_length)
input_data = np.random.rand(*input_shape).astype(np.int64)

# Perform inference and measure throughput
num_iterations = 100  # Number of inference iterations
start_time = time.time()
for _ in range(num_iterations):
    output = ort_session.run([], {"input.1": input_data})  # Replace "input" with the actual input name in yourmodel
end_time = time.time()

# Calculate throughput
total_time = end_time - start_time
average_time_per_iteration = total_time / num_iterations
throughput = 1 / average_time_per_iteration  # Throughput in inferences per second

print(f"Throughput: {throughput} inferences/second")
```

Plotting graph of latency vs input size code:

```python
import matplotlib.pyplot as plt

input_sizes = [1,2,4,6,8,12]  # Example input sizes (batch size)
latencies = []

for batch_size in input_sizes:
    start_time = time.time()
    output =  ort_session.run(None, { ort_session.get_inputs()[0].name: input_ids})
    end_time = time.time()
    latency_ms = (end_time - start_time) * 1000
    latencies.append(latency_ms)

# Plot latency vs. input size
plt.plot(input_sizes, latencies, marker='o')
plt.xlabel("length")
plt.ylabel("Latency (ms)")
plt.title("ONNX Model Latency vs. Input Size")
plt.grid(True)
plt.show()
```

Another Hugging Face model(Tiny bert) and it's ONNX model implementation code:

```python
import torch
from transformers import BertTokenizer, BertForSequenceClassification
import time
# Load the TinyBERT model and tokenizer
model_name = "huawei-noah/TinyBERT_General_4L_312D"
tokenizer = BertTokenizer.from_pretrained(model_name)
model = BertForSequenceClassification.from_pretrained(model_name)

# Sample input text (replace with your own text)
input_text = "what is the capital of india?"
length=len(input_text)
print(f"Length of the string: {length}")

# Tokenize input text
input_ids = tokenizer(input_text,padding=True, truncation=True, return_tensors="pt")["input_ids"]

# Measure latency
with torch.no_grad():
    start_time = time.time()
    outputs = model(input_ids)
    end_time = time.time()

latency = (end_time - start_time) * 1000  # Convert to milliseconds

print("Latency:", latency, "ms")
for _ in range(3):
     model(input_ids=input_ids)
# Measure throughput
start_time = time.time()
num_exam=100
for _ in range(num_exam):  # Adjust the number of iterations according to your requirements
    model(input_ids=input_ids)
end_time = time.time()
total_time = end_time - start_time
throughput = num_exam / total_time
print("Throughput:", throughput, "inferences/second")
```

```python
import onnxruntime
import numpy as np
from transformers import BertTokenizer
 # Load the ONNX model
onnx_model_path = "./quantized_tinybert (1).onnx"
ort_session = onnxruntime.InferenceSession(onnx_model_path)
 # Tokenize inp
tokenizer = BertTokenizer.from_pretrained('huawei-noah/TinyBERT_General_4L_312D')
input_text = "20 * 5 + 3 = "
length=len(input_text)
print(f"Length of the string: {length}")
input_ids = tokenizer.encode(input_text, return_tensors="np", padding=True,
truncation=True)
input_ids = np.array(input_ids, dtype=np.int64)
 # Run the model
ort_inputs = {"input_ids": input_ids}
ort_outs = ort_session.run(None, ort_inputs)
  # Get the prediction
predicted_class = np.argmax(ort_outs[0])
print("Predicted class:", predicted_class)
import time
start_time = time.time()
output = ort_session.run(None, {ort_session.get_inputs()[0].name:  input_ids})
end_time = time.time()
# Calculate latency
latency_ms = (end_time-start_time) *1000
print(f"Latency: {latency_ms:.2f} ms")
```

```python
# Generate input data (you need to replace this with your actual input data)
input_shape = (1, 8)  # Assuming input shape is (batch_size, sequence_length)
input_data = np.random.rand(*input_shape).astype(np.int64)

# Perform inference and measure throughput
num_iterations = 100  # Number of inference iterations
start_time = time.time()
for _ in range(num_iterations):
    output = ort_session.run([], {"input_ids": input_data}) # Replace "input" with the actual inputname in yourmodel
end_time = time.time()

# Calculate throughput
total_time = end_time - start_time
average_time_per_iteration = total_time / num_iterations
throughput = 1 / average_time_per_iteration  # Throughput in inferences per second

print(f"Throughput: {throughput} inferences/second")
```

Plotting graph of latency vs input size code:

```
import matplotlib.pyplot as plt

input_sizes = [1,2,4,6,8,12]  # Example input sizes (batch size)
latencies = []

for batch_size in input_sizes:
    start_time = time.time()
    output =  ort_session.run(None, { ort_session.get_inputs()[0].name: input_ids})
    end_time = time.time()
    latency_ms = (end_time - start_time) * 1000
    latencies.append(latency_ms)

# Plot latency vs. input size
plt.plot(input_sizes, latencies, marker='o')
plt.xlabel("length")
plt.ylabel("Latency (ms)")
plt.title("ONNX Model Latency vs. Input Size")
plt.grid(True)
plt.show()
```

9. Result Analysis:

CPU performance:

Input 1: ”What is capital of India?”

Length of the string:25

Model	latency	throughput
Tinybert(hugging face)	177.28829383850098 ms	69.49934109656841 inferences/second
Tinybert(ONNX)	2.58 m s	405.21309340479917 inferences/second
Sentimental analysis model(hugging face)	281.29 ms	4.53 examples/second
Sentimental analysis model(ONNX) (Input: ”What capital of India?”)	22.03 ms	26.907246461045794 inferences/second

Input 2: ”20 * 5 + 3 = ”

Length of the string:13

Model	latency	throughput
Tinybert(hugging face)	232.57684707641602 ms	4.947033484118066 inferences/second
Tinybert(ONNX)	2.44ms	535.9578113958811 inferences/second
Sentimental analysis model(hugging face)	232.16ms	4.25 examples/second
Sentimental analysis model(ONNX) (input: ”20 * 5 = ?”)	32.34 ms	19.217632388847985 inferences/second

NLP-graph:

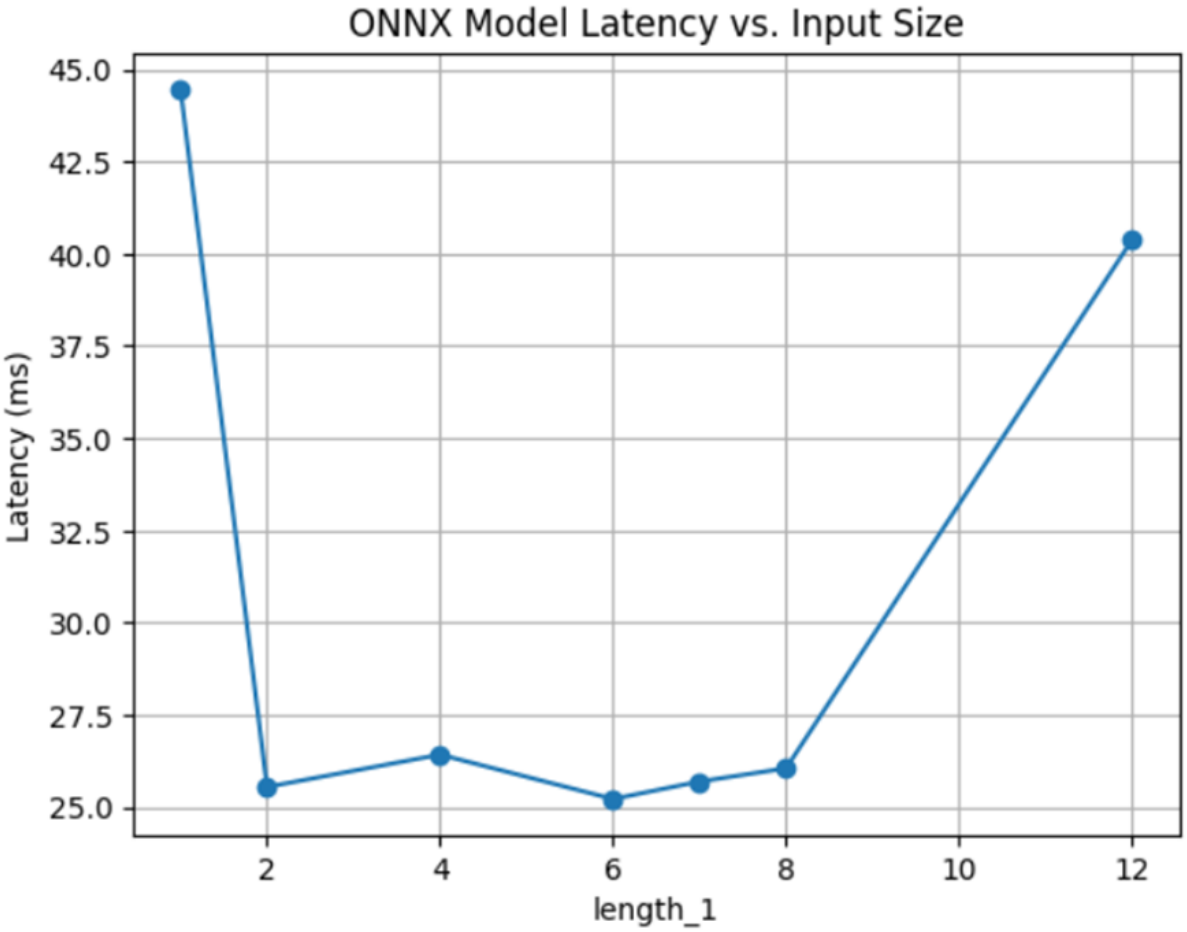


Figure 9.1: NLP MODEL

Tiny bert graph:

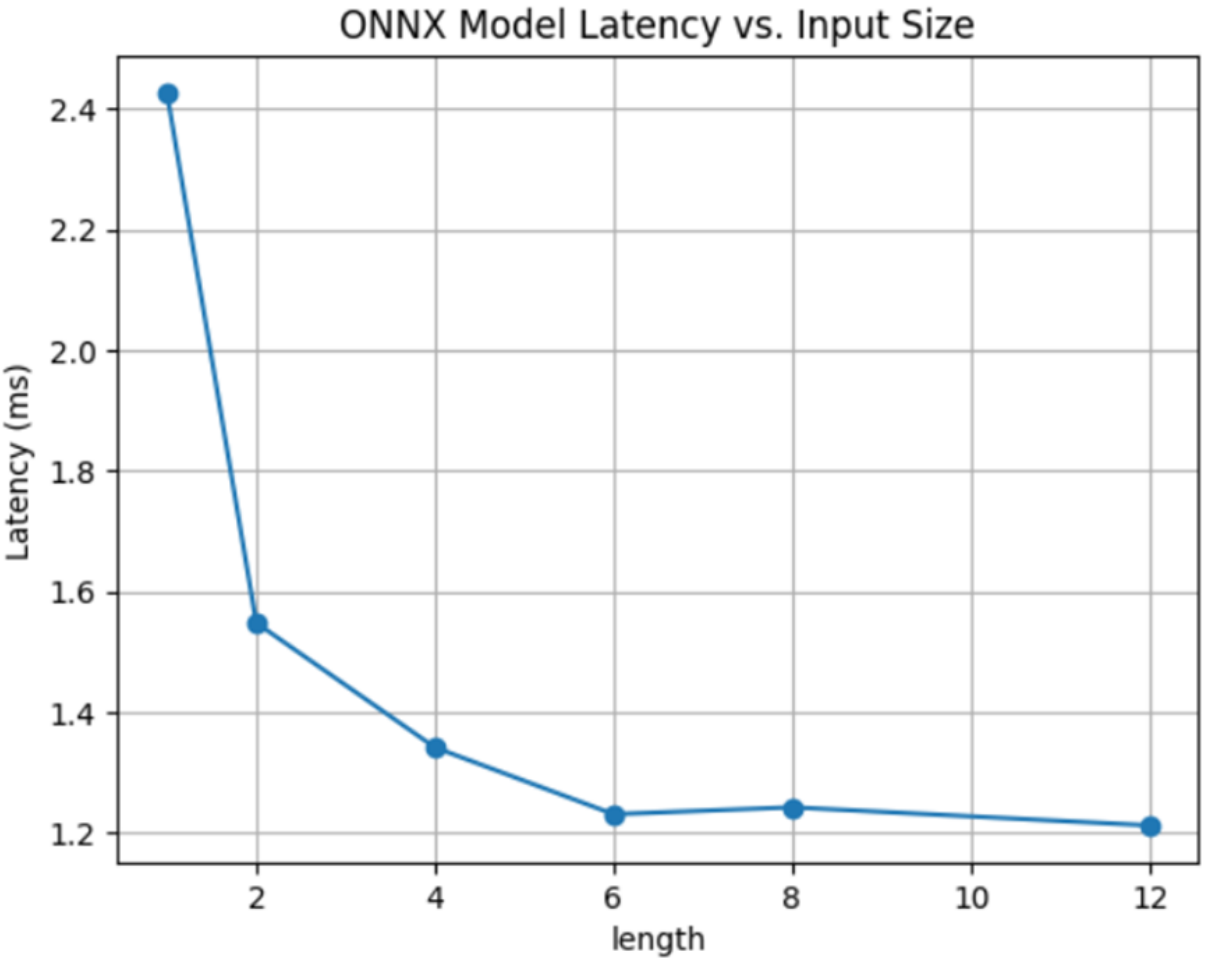


Figure 9.2: TINY BERT MODEL

GPU performance:

Input 1: ”What is capital of India?”

Length of the string:25

Model	latency	throughput
Tinybert(hugging face)	17.5778865814209 ms	47.60965005130096 inferences/second
Tinybert(ONNX)	1.96 ms	635.5063833913642 inferences/second
Sentimental analysis model(hugging face)	117.47 ms	8.29 examples/second
Sentimental analysis model(ONNX) (Input: ”What capital of India?”)	13.83 ms	35.132764414781164 inferences/second

Input 5: ”20 * 5 + 3 = ”

Length of the string:13

Model	latency	throughput
Tinybert(hugging face)	5.8956146240234375 ms	100.99905726597501 inferences/second
Tinybert(ONNX)	1.69 ms	629.9304933947699 inferences/second
Sentimental analysis model(hugging face)	123.08 ms	8.48 examples/second
Sentimental analysis model(ONNX) (input: ”20 * 5 = ?”)	10.99 ms	60.32213192935143 inferences/second

10. FUTURE SCOPE:

This project can be expanded in the future to investigate new hardware technologies like implenting on FPGA and to improve the deployment of large language models (LLMs), such as quantum computing and neuromorphic computing. Furthermore, researching hybrid architectures—which integrate CPUs, GPUs, and specialized accelerators—may result in systems that are better suited for the implementation of LLM. The project could be expanded to include edge computing environments, automated optimization methods, security and privacy issues, industry-specific deployment strategies, and an investigation of the moral and societal ramifications of implementing LLMs on various hardware platforms. The efficiency, scalability, and appropriate deployment of LLMs in a variety of applications are the goals of these future area.

11. CONCLUSION

In this study, we examined how well Large Language Models (LLMs) performed when implemented on CPUs, GPUs. We set out to identify the best hardware platform based on energy efficiency, cost, and other considerations, in order to achieve efficient deployment.

CPU Performance: Although general-purpose computers are dominated by CPUs, which are flexible devices, they are not able to meet the high computational needs of language learning modules. Performance is slower and energy consumption is higher when using CPUs because they have a restricted capacity for parallel processing when compared to GPUs and Estimation on FPGAs.

GPU speed: GPUs offer a notable speed advantage over CPUs due to their high parallelism and matrix operation-optimized architectures.Their efficiency in handling large-scale computations makes them popular for training and deploying LLMs. GPUs aren't necessarily the most economical option for all deployment scenarios, either, as they can be power-hungry.

Estimation on FPGA Performance: Based on our research, FPGAs provide the optimum combination of cost, energy efficiency, and performance when it comes to deploying long-lived machines. Customizable hardware configurations made possible by FPGAs provide optimization for certain LLM workloads. This leads to reduced power usage and quicker inference times as compared to CPUs and GPUs. Moreover, future-proofing is facilitated by the reconfigurability of FPGAs when models and algorithms change.

Last Word of Advice:

We advise using FPGA-based solutions for installing LLMs due to the better performance and energy economy that they have shown, particularly in situations where power consumption and operating expenses are crucial factors. Due to their superior raw computing capacity, GPUs are still a great option for initial model training; however, FPGAs perform well in deployment situations, making them the best option for operating LLMs in production.

Businesses can realize substantial speed and efficiency gains by utilizing FPGAs, which will allow LLMs to reach their full potential across a range of applications. But we can say this advice based on estimation of FPGA Computational Power.

12. REFERENCES:

1. A Survey on Hardware Accelerators for Large Language Models (arxiv.org)
2. FlightLLM: Efficient Large Language Model Inference with a Complete Mapping Flow on different Hardware: 2401.03868 (arxiv.org)
3. A Study on the Implementation of Generative AI Services Using an Enterprise Data-Based LLM Application Architecture. arXiv preprint arXiv:2309.01105 (2023).
4. Transformers and LLMs: The Next Frontier in AI :https://arxiv.org/pdf/2401.03868 (linkedin.com)
5. https://arxiv.org/pdf/1706.03762.pdf
6. https://www.databricks.com/blog/us-air-force-hackathon-how-large-language-models-will-revolutionize-usaf-flight-test
7. https://www.achronix.com/blog/fpga-accelerated-large-language-models-used-chatgpt
8. https://www.aldec.com/en/company/blog/167–fpgas-vs-gpus-for-machine-learning-applications-which-one-is-better
9. To export a model : https://huggingface.co/docs/optimum/exporters/onnx/usageguides/export
10. transformers:https://huggingface.co/docs/transformers/serializationonnx
11. 2406.01698 (arxiv.org)
12. Han, S., et al. (2016). ”Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding.”
13. Shoeybi, M., et al. (2019). ”Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism.”