OULUN AMMATTIKORKEAKOULU

Aki Korvala

Analysis of LLM-models in optimizing and designing VHDL code

Master Thesis

Analysis of LLM-models in optimizing and designing VHDL code

Master thesis

Aki Korvala
Master degree
Autum 2023
Modern SW and Computing technol-
gies
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Master degree, Modern Software and Computing Techniques

Author: Aki Korvala
Title of the thesis: Analysis of LLM-models in optimizing and designing VHDL code
Thesis examiner: Ville Majava
Term and year of thesis completion: Autum 2023
Pages: 44 + 4 appendices

In the evolving world of software and hardware co-design, VHDL has emerged as a pivotal language for hardware description. The objective of this research was to harness the capabilities of Large Language Models (LLM) to optimize and streamline VHDL code design. The underlying premise was rooted in the belief that feeding vast amounts of raw VHDL data to these models would enable them to autonomously learn and generate efficient VHDL code snippets.

This study adopted a systematic approach involving the fine-tuning of pre-existing LLMs, specifically focusing on model adaptations for VHDL code generation. The methodology encompassed iterative processes, characterized by several adjustments to parameters, data parallelism leveraging, and base model load configurations. Additionally, considerations like unsupervised versus supervised learning approaches were evaluated, underscoring the pivotal role of labeled data for targeted outcomes.

The empirical findings shed light on the significance of data quality over quantity. While the initial approach embraced a 'feed and learn' attitude, it became evident that for fine-tuning purposes, curated and labeled data was paramount. Testing of the refined models, against a set of basic to advanced VHDL challenges, showcased the potential of the LLMs, with ChatGPT models outperforming others considerably.

In conclusion, while the journey unveiled the intricacies of training and fine-tuning LLMs for VHDL, it also highlighted the invaluable role of data. Future endeavors in this domain can delve deeper into supervised training methods, base model variations, and the intricate interplay of quantization during training. The research accentuates the promise of LLMs in the domain of VHDL design, steering a path for more refined and nuanced implementations in the future.

Keywords: VHDL, Large Language Models, Machine Learning, Python, GPU, LoRA, Codegen25

CONTENTS

1 INTRODUCTION

1.1 Background and Motivation

In the middle of all the excitement about Artificial Intelligence and Machine Learning, people are wondering how it can be used in different areas in technology.
These are not just buzzwords; they are powerful tools that are reshaping how we approach problems in many areas of life. One of the intriguing questions arising is how we can harness this power for creating a VHDL code and further, especially for FPGA (Field Programmable Gate Array)/SOC(System On Chip) creations.

This thesis is our exploration into that question. We are not just theorizing; we are actively investigating the practicality of using machine learning to aid in VHDL code creation. By delving into this, we aim to understand the potential benefits, the challenges, and the overall impact on our design processes. The primary focus is on how machine learning can assist in generating VHDL code more efficiently and effectively. While we are excited about the broader implications and future possibilities, this document will be centred on the core topic of VHDL code generation with the help of machine learning and further studies are out of scope of this thesis.

1.2 Objectives of the Study

Given the promising capabilities of Artificial Intelligence and Machine Learning in the realm of VHDL code generation, this study is anchored on the following specific objectives:

- Test Case Development: Prior to delving into the intricacies of the models, our primary aim is to curate a set of approximately 10 test cases. These will act as the foundational benchmarks to evaluate the efficacy of various LLM models in generating VHDL code.

- Custom Model Creation: While numerous LLM models are accessible, our ambition is to adapt an open-source model specifically for VHDL code generation. This tailored approach is anticipated to address the distinct intricacies and demands of VHDL.

- Comparative Analysis: We will not be relying on just our custom model. The study will rigorously test the VHDL code outputs from three distinct models: the widely recognized (Open AI, 2023), our own model, and with basemodel, Codegen25.
- Iterative Improvement: Post-analysis, depending on the insights we gather, there is an objective to refine our custom model further. The idea is to ensure it is not just functional but excels in VHDL code generation.

By the end of this study, we aim to have a clearer understanding of the role LLMs can play in VHDL code creation and the efficacy of our custom model, and the potential roadmap for future enhancements.

1.3 Scope of the Thesis

In this research, we embark on a comprehensive exploration of the role and potential of Large Language Models (LLMs) in the domain of VHDL code creation. The journey begins with a foundational understanding of LLMs, shedding light on their architecture and positioning within the broader landscape of machine learning. This foundational knowledge is complemented by an overview of the essential phases in machine learning, from the initial stages of data acquisition to the final steps of model deployment.

A significant portion of our research is dedicated to the evaluation criteria for LLMs in VHDL code generation. Ensuring the integrity and quality of our VHDL data is paramount. Thus, we undertake rigorous processes of data standardization, cleaning, and organization. This data is then strategically segmented into distinct sets for training, validation, and testing purposes.

Our choice of a specific open-source LLM as a foundational model is not arbitrary. We delve into the rationale behind this selection, outlining our vision for adapting and customizing it to our unique requirements. As we progress, the methodologies employed in model customization, training, and fine-tuning are elucidated, along with the techniques adopted for model validation and performance evaluation.

The culmination of our research efforts will be a systematic evaluation of three distinct LLMs, gauging their efficacy in VHDL code generation. However, it is pertinent to note certain boundaries in our study. While we strive for a comprehensive exploration, aspects such as human-assisted model fine-tuning and real-world model deployment are beyond the immediate purview of this thesis, presenting potential avenues for future research endeavours.

2 LITERATURE REVIEW

2.1 Overview of VHDL Code Design

VHDL, standing for VHSIC Hardware Description Language, is a pivotal tool in the world of digital design. Think of it as a language that tells a computer exactly how to create a digital circuit.
In the similar manner as an architect uses blueprints to design a building, VHDL provides a detailed plan for digital circuits.

For those familiar with programming, VHDL might resemble a traditional coding language. Here is a brief example:

```
-- Description: This file contains a mix of VHDL code, comments, errors, and documentation.


-- LIBRARY DECLARATION
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL:


-- ENTITY DECLARATION
entity Main is
    Port (
        clk    : in STD_LOGIC;
        reset  : in STD_LOGIC;
        input  : in UNSIGNED (7 downto 0);
        output : out UNSIGNED (7 downto 0)
    );
end Main;


-- ARCHITECTURE DEFINITION
architecture Behavioral of Main is
begin
    process(clk, reset)
    begin
        if reset = '1' then
```

```
        output <= (others => '0');
    elsif rising_edge(clk) then
        output <= input; -- Direct assignment is fine since both are UNSIGNED now
    end if;
  end process;
end Behavioral;
```

This is a basic VHDL code that describes an acts as a register that can be reset, storing and outputting the input data on each clock cycle

The way VHDL is structured ensures that there is no room for guesswork. Every part of the digital design is clearly defined using elements like entities, architectures, and processes. However, there is a notable challenge: VHDL, despite its capabilities, presents difficulties in the availability of pre-existing code repositories. On platforms such as GitHub, where developers typically share and collaborate on code, VHDL contributions are relatively scarce. This scarcity can be attributed to the proprietary nature of VHDL designs within companies, where they are often safeguarded as trade secrets.

For our research, we are not just relying on what is out there. We are diving deep, creating our own models using unique VHDL data. This hands-on approach ensures that our work is both innovative and tailored to our specific needs.

2.2 Introduction to Large Language Models (LLMs)

In the ever-evolving landscape of artificial intelligence, Large Language Models (LLMs) have emerged as a transformative force. The journey of LLMs began with simpler models that could predict the next word in a sequence. Over time, as computational power increased and datasets grew, these models evolved to understand context, nuances, and even generate coherent paragraphs of text. (al, 2003)

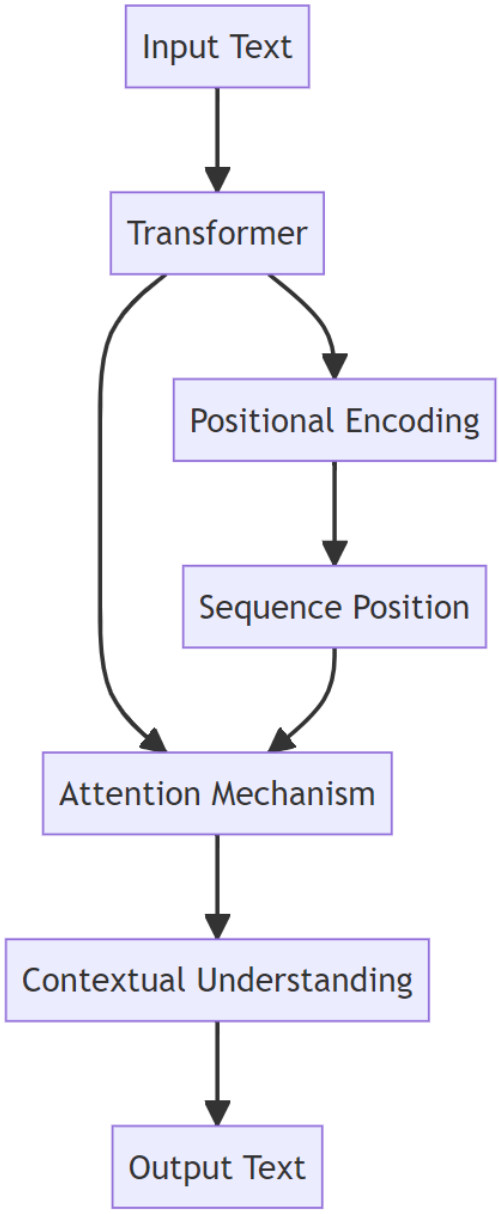
The last decade has witnessed significant advancements in this domain. From the initial RNNs (Recurrent Neural Networks) and LSTMs (Long Short-Term Memory networks) that could remember past information in sequences, the field transitioned to the development of transformers, which

brought about a paradigm shift. Introduced the transformer architecture laid the foundation for models like BERT, GPT, and their subsequent iterations. (all A. V., 2023),

Today, LLMs like OpenAI's GPT-3 or Google's BERT have become household names in the tech community. They are not just limited to predicting text but have applications ranging from code generation to answering complex questions, translating languages, and even creating art. Their ability to understand and generate human-like text has opened up avenues previously thought impossible. Note that these are not usable for this thesis as these models are under licence.

LLMs are like super-smart systems that can read, understand, and even write text almost like humans do. These models are trained using vast amounts of data, and their primary function is to predict the next word in a sequence, enabling them to craft coherent sentences. One of the main reasons LLMs excel is due to the transformer architecture. This structure allows them to consider entire sentences, leading to more nuanced and accurate predictions.

Now, let us visualize how transformers work:



Transformers are like the brainpower behind modern language models. Imagine you are reading a book, and as you read each word, you are constantly thinking about the words that came before it to understand the full story. That is what transformers do, but for machines. They help models look at entire sentences or even paragraphs to understand the context and make better predictions. (all A. V., 2023)

How Transformers Power LLMs:

Input Text: This is where it all begins. We give the model some text to start with.
Transformer: Picture this as the heart of the operation. It takes in the input and gets to work.

Positional Encoding: Words have a specific order in sentences, and this step ensures the model recognizes that. It is all about understanding the sequence and flow of words.
Sequence Positioning: After understanding the order of words, the model also gauges the significance of each word's position in a sequence, ensuring that the context is preserved.
Attention Mechanism: This is where the magic happens. It is like the model's way of highlighting key words in a sentence, determining which ones hold more weight.
Contextual Understanding: After all the processing, the model gets a clear picture of the text's context, ensuring its responses are on point.
Output Text: With all the information processed, the model provides its output. This could be similar to the input, a concise summary, or any other relevant responses.

And here's a bonus: Through prompt engineering, we can guide the model's output, fine-tuning its responses to align with our needs. It is like having a conversation with the model, steering it in the direction we want.

2.3 Existing LLMs in VHDL Code Optimization

The application of Large Language Models in VHDL code optimization is still in its nascent stages, but there are a few notable models that have made strides in this domain. Codegen16 with Verilog is one such model.
It is a product of extensive training on diverse datasets, primarily sourced from GitHub repositories and academic PDFs. While the foundational Codegen model it is not inherently equipped with VHDL or Verilog expertise, its Verilog variant does possess some knowledge, albeit a minimal 0.3%. (all S. T., 2022)

Following Codegen16, the Codegen25emerged as a more advanced successor. Developed using the robust StarCoderData (contact@bigcode-project.org, 2023), this model boasts a slightly enhanced VHDL proficiency, capturing about 1.3% of VHDL knowledge from GitHub. This increment, though seemingly modest, represents a significant leap in the model's capability to understand and generate VHDL code. (all E. N., 2023)

The evolution of Codegen models underscores the rapid advancements in the field. With each iteration, these models are becoming more adept at understanding the intricacies of VHDL and Verilog, paving the way for more efficient code generation and optimization.

However, a pertinent question arises: Is it prudent to adopt a model that hasn't been extensively trained with VHDL or Verilog? The answer it is not straightforward. While these models offer a foundation, their efficacy in VHDL code generation might be limited due to the scarcity of VHDL training data. For organizations and individuals looking to harness the power of LLMs for VHDL optimization, it might be beneficial to consider models that have undergone specialized training in VHDL or even contemplate custom training tailored to specific needs. In this thesis we will use Codegen25 as a basis for fine-tuning and also Codegen25 will be one of three LLM-model which is used in test case testing.

2.4 Limitations of Current Approaches

While the previous sections highlighted the potential and advancements of LLMs in VHDL code optimization, it is essential to address the challenges and limitations inherent to current methodologies. A predominant hurdle is the scarcity of VHDL training data. For LLMs to be proficient, they require vast amounts of data for training. However, the VHDL domain presents a unique challenge in this regard.

Most of the VHDL data used for fine-tuning these models is sourced from platforms like GitHub and academic research. This data, though valuable, is limited in volume. The primary reason for this limitation is the proprietary nature of VHDL code. Unlike many open-source programming languages that developers freely share and collaborate on, VHDL, especially in the context of SOC (System On Chip) and FPGA designs, is often proprietary. Companies and institutions tend to guard their VHDL designs, preventing them from being publicly accessible.

This restricted access to VHDL data poses a significant challenge. Without ample data, the efficacy of LLMs in generating and optimizing VHDL code is inherently capped. They might not capture the nuances and intricacies of complex VHDL designs, leading to suboptimal code generation.

However, in the context of this thesis, this limitation is circumvented. Our approach involves leveraging proprietary VHDL code for training, ensuring that the models are exposed to a rich and diverse dataset, enhancing their proficiency in VHDL code generation and optimization.

2.5 The Dilemma of Licensed Data in LLMs

In the context of training Large Language Models (LLMs), the process involves presenting these models with extensive corpora of text, which enables them to discern linguistic patterns, contextual nuances, and additional language-related complexities.

- **Source of Data**: Many LLMs are trained on data scraped from the internet, which includes websites, books, articles, and more. Some of this content is open-source or in the public domain, but a significant portion is copyrighted or licensed.

- **Legal Implications**: Using copyrighted or licensed data without permission for training can lead to legal challenges. Content creators or publishers might argue that their intellectual property rights are being infringed upon.

- **Ethical Concerns**: Beyond the legal aspect, there is an ethical dimension. If an LLM is commercialized and profits from its capabilities, is it fair if it was trained on data that was not freely available or was used without proper attribution?

- **Transparency and Trust**: For users, knowing that an LLM might generate outputs based on copyrighted data can lead to trust issues. If the model suggests a piece of text or code, how can users be sure it is original and not something replicated from a licensed source?

Potential Solutions:

- **Data Licensing:** Before using any data for training, AI developers can ensure they have the right licenses or permissions.
- **Synthetic Data**: Using artificially generated data that does not infringe on copyrights.

- **Open-Source Collaboration**: Encouraging more open-source contributions can provide LLMs with a wealth of data that's freely available and does not come with legal strings attached.

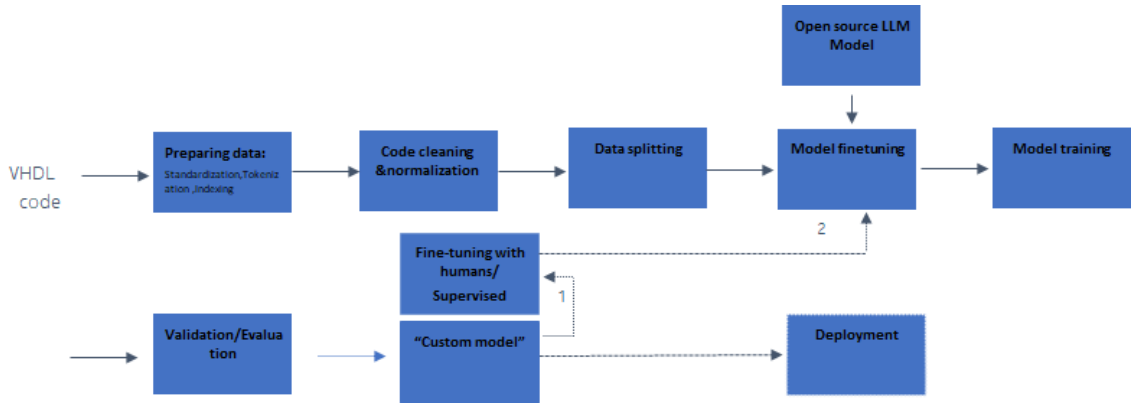
The strategy for this thesis includes selecting a model that is licensed under the MIT License or the Apache License, both of which permit the use and modification of the software, thereby enabling the retraining of a custom model.

3 METHODOLOGY FOR CUSTOM MODEL CREATION

This section delves into the comprehensive methodology for devising custom models tailored for VHDL code generation. We dissect the procedure into its core components, offering a lucid breakdown of each phase.

3.1 Custom model creation process

Example of custom model creation



Crafting a custom model is a multifaceted endeavour, generally segmented into three primary steps:

- **Open Source Model Selection:** Initiating the process requires picking a suitable open-source model that seamlessly integrates with VHDL code generation requirements. This model acts as the bedrock upon which future alterations are based.

- **Data Transformation**: The subsequent phase is dedicated to data refinement. Key processes involved are data standardization, tokenization, and code sanitization. The main aim is to mold the data for seamless integration into model training and enhancements.

- **Model Refinement**: The data is compartmentalized into training, test, and validation subsets. Post this, the model is optimized using the training subset. The model's efficacy in

VHDL code generation is assessed based on its performance on the test and validation sets.

Note: It should be however noted that this discourse will not extend to model deployment.

3.2 Selection of Open Source Model

The process of selecting an appropriate Large Language Model (LLM) for custom model creation is crucial. It is not just about choosing the most popular or the latest model; it is about understanding the specific needs of the project and aligning them with the capabilities of the LLM. Here are some detailed considerations:

- **Model Size:** The size of the model often correlates with its capabilities. Larger models tend to be more powerful, but they also require more computational resources. It is essential to strike a balance between the model's size and the available infrastructure.

- **Training Data**: Understanding the data on which the LLM was trained is vital. It gives insights into the model's strengths and potential biases. For VHDL code generation, a model trained on a diverse set of programming languages might be more beneficial than one trained solely on literary texts.

- **Prompting Response Time:** The speed at which the model provides outputs is crucial, especially if it is used in real-time applications. A faster response time enhances user experience but might come at the cost of detailed outputs.

- **Fine-tuning Capabilities**: Not all LLMs allow for fine-tuning. Choosing a model that supports fine-tuning is essential if there is a need to adapt the model to specific VHDL datasets or requirements.

- **Deployment Costs**: Deploying and maintaining LLMs can be resource intensive. It is crucial to factor in the costs associated with hosting, computation, and potential updates.

- **Flexibility in Integration**: The world of LLMs is rapidly evolving. New models and architectures are being introduced frequently. Therefore, when building the integration code, it is wise to ensure flexibility. This means structuring the code in a way that allows for easy swapping or updating of the underlying LLM without extensive modifications.

In essence, the selection process is a blend of technical evaluation and strategic foresight. Given the rapid advancements in the field of LLMs, it is not just about what works best now, but also about ensuring adaptability for future innovations. By considering the factors mentioned above and continuously monitoring the landscape of LLMs, one can make informed decisions that align with both current and future project needs.

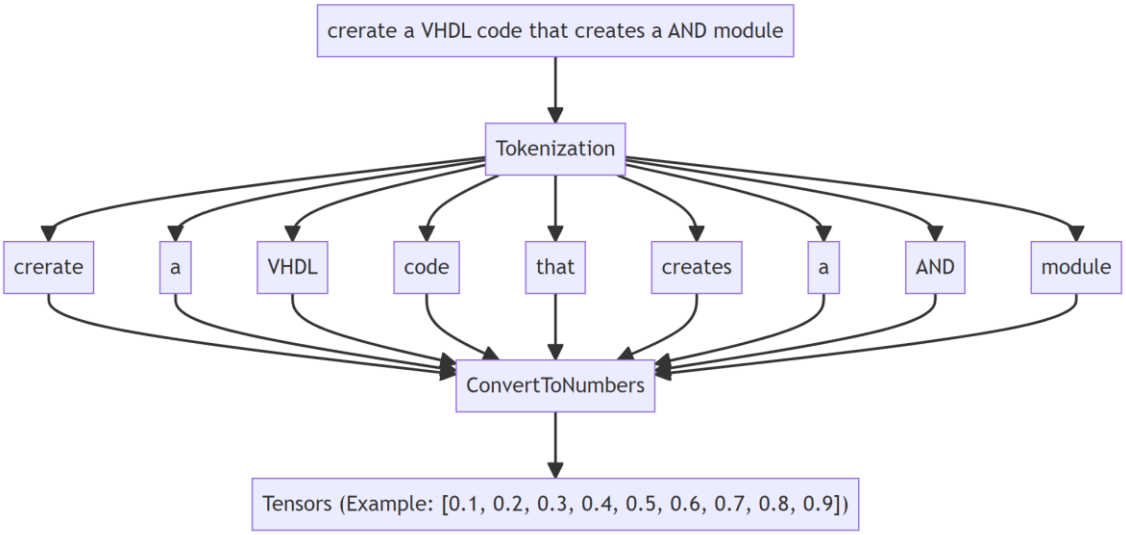
3.3 Data Collection and VHDL Code Preparation

For the successful application of LLMs to VHDL code, preprocessing the code into a format that the chosen model can understand is paramount. Raw text or VHDL code, in its native form, is not directly interpretable by LLMs. Such inputs must be numerically represented and structured into tensors. While this transformation is facilitated automatically by specific code libraries and transformers, understanding the underlying process is essential.

Supervised Learning and Data Labeling: Supervised learning is a type of machine learning paradigm where the model is trained on labeled training data, meaning that each example in the dataset is paired with the correct output. The model learns from this data and then makes predictions based on it. Data labeling, in this context, refers to the process of tagging data with the correct answer or label. For instance, in a classification task, each input data point will be labeled with its correct class or category.

Difference to Unsupervised Learning and Fine-Tuning: Unsupervised learning, on the other hand, involves training the model on data without explicit labels. The model tries to learn the underlying structure from the data without any guidance. When it comes to fine-tuning, especially in the context of LLMs, it often involves supervised learning. This is because, during fine-tuning, one typically labels a smaller subset of data to guide the model towards a specific task, rather than feeding all the data in a big chunk to the base model.

A critical step in this process is determining the size of the file or data chunk intended for the model. When working with LLMs, tokenization is the preferred method for processing input data, especially textual data. Tokenization involves breaking down text into smaller units, or tokens. These tokens are then numerically represented, forming the basis for tensors that serve as model input.



Example of input prompt" Create a VHDL code that …"

Tokenization assigns a unique ID to each token, effectively converting it into a number. Here's a practical example of tokenization using a Hugginface library:

from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained("Salesforce/codegen25-7b-multi")
 # Passing text to the tokenizer
encoded_input = tokenizer("Do not meddle in the affairs of wizards, for they are subtle and quick to anger.")
print(encoded_input)
Output:
`{'input_ids': [101, 2079, 2025, 19960, 10362, 1999, 1996, 3821, 1997, 16657, 1010, 2005, 2027, 2024, 11259, 1998, 4248, 2000, 4963, 1012, 102],`

```
 'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

The tokenizer returns a dictionary with three important items:

input_ids: These are the indices corresponding to each token in the sentence.
attention_mask: This indicates which tokens should be considered during model processing.
token_type_ids: This identifies the sequence a token belongs to, especially useful when dealing with multiple sequence.

In essence, data preparation is about collecting and refining the data to fine-tune the LLM. It is vital to ensure that this data is representative of the intended task. Techniques like data augmentation can also be employed to enhance the diversity and volume of training data. Proper data preparation lays the foundation for the model to learn task-specific patterns and nuances effectively.

3.4 Finetuning(training) Process

The landscape of Large Language Models (LLMs) is vast, with hundreds of models available for various purposes. Some are licensed for specific uses, while others are open for educational or business applications. Given our objective to employ an LLM tailored for code generation, our primary criterion should be the model's proficiency in understanding syntax, be it C or VHDL. Standard LLMs might not inherently understand these languages if they haven't been trained with such code. Given the scarcity of LLMs trained with VHDL, our approach necessitates fine-tuning our chosen model with new datasets, in this case, VHDL code in text format. The ultimate goal is to optimize the model for specific applications, as illustrated in our test cases. Essentially, fine-tuning adjusts the model's parameters to align with the distribution of the new dataset.

Several techniques exist for data training, including data synthesis, continual learning, transfer learning, one-shot learning, few-shot learning, and multi-task learning. The choice of technique depends on the application and the available data.

3.4.1 Different LLM Fine-tuning Techniques

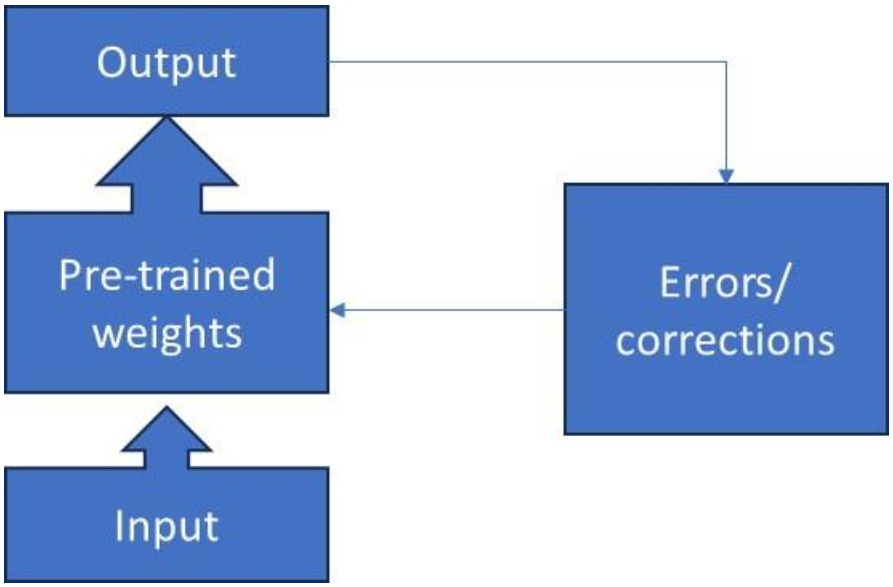
Fine-tuning techniques vary in their application and effectiveness. In some scenarios, the goal is to repurpose a model for a different task. For instance, an LLM pre-trained for text generation might be adapted for sentiment or topic classification. This repurposing involves leveraging the embeddings produced by the transformer component of the model. These embeddings are numerical representations capturing various features of the input. While some LLMs directly generate embeddings, others, like the GPT family, use embeddings to produce tokens or text.

Repurposing typically involves connecting the model's embedding layer to a classifier (e.g., a set of fully connected layers) that maps the embeddings to class probabilities. In this setup, only the classifier is trained on the embeddings, with the LLM's attention layers remaining static. This approach is computationally efficient but requires a supervised dataset comprising text examples and their corresponding classes.

However, in situations where updating the transformer model's parameter weights is essential, full fine-tuning becomes necessary. This process, which can be computationally intensive, involves unfreezing the attention layers and updating the entire model. Techniques like Parameter-efficient fine-tuning (PEFT) with LoRA can make this process more efficient by representing weight updates with smaller matrices through low-rank decomposition. With LoRA,one can fine tune LLM´s with a fraction of the cots it would normally take(i.e time and GPU power). (all E. H., 2021)
At the core, every LLM is structured as a transformer model, comprising several layer blocks, each housing learnable parameters. The fine-tuning process picks up from where the pre-training concluded. The model ingests input from the fine-tuning dataset, predicts subsequent tokens, and juxtaposes its output with the ground truth. This iterative process of adjusting weights to align predictions with actual data fine-tunes the LLM for the targeted task.

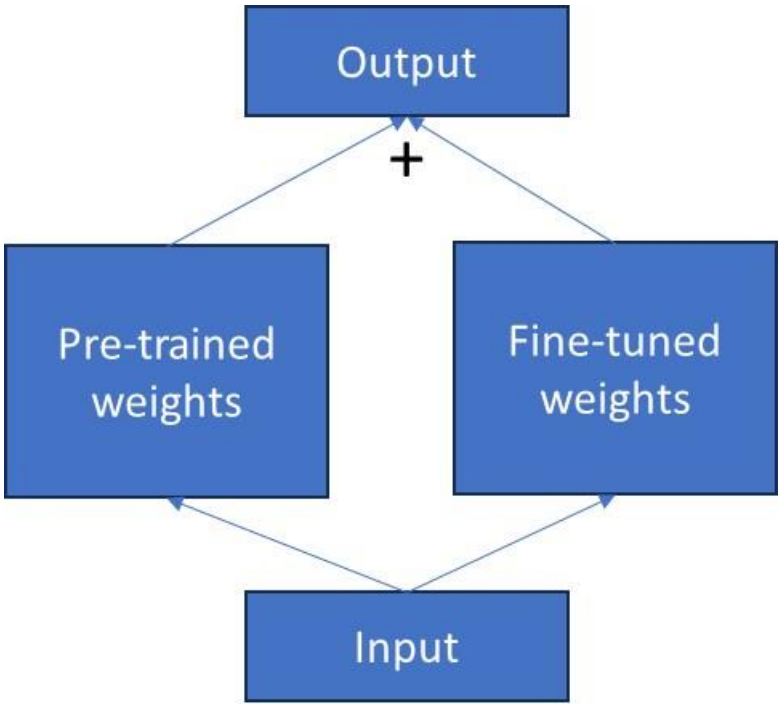
3.4.2 Classic LLM Fine-Tuning vs. Modified Approach

In the traditional approach, the fine-tuning process involves tweaking the original weights of the model to enhance its performance for a specific task. This is an iterative task which consumes time and capacity and lasts until we have a model ready.



Classical ML model fine-tuning

However, a modified approach proposes freezing the original weights during the fine-tuning phase. Instead, adjustments are made to a separate set of weights, termed as "fine-tuned" weights, which are then added to the original parameters, distinguishing them as "pre-trained" and "fine-tuned" weights.



Initially,the pre-trained weights are fixed, and subsequently, only the weights that are subject to fine-tuning are adjusted and summed.

3.4.3 Low-Rank Adaptation (LoRA)

Before we delve into LoRA, it is crucial to understand that our model parameters are essentially expansive grids or matrices that form a complex network, representing the vast landscape of language nuances. These matrices have a certain "rank," which signifies the number of unique pathways or columns present in them. (all E. H., 2021),

LoRA, a technique pioneered by researchers at Microsoft, proposes a more efficient approach to fine-tuning Large Language Models (LLMs) for specific tasks. Instead of adjusting the entire matrix, which is characteristic of open-source LLMs, it suggests that we can retain a significant portion of the model's learning capacity by only modifying a smaller section of these matrices, thereby reducing the dimensions of the downstream parameters. This facilitates a clear separation between the pre-trained weights (inherited from the open-source LLM) and the fine-tuned weights that we develop during the fine-tuning process.

In essence, LoRA introduces two new sets of matrices during the fine-tuning process: one that transitions the input parameters from their original state to a more focused, low-rank dimension, and another that translates this low-rank data back to the output dimensions of the original model. This approach significantly reduces the computational demands and associated costs of full fine-tuning, as the modifications are limited to the LoRA parameters, which are considerably fewer than the original weights.

3.4.4 Enhancements with LoRA

Despite the memory overhead incurred due to the separation of pre-trained and fine-tuned weights, LoRA presents an avenue for further improvements. The process of adding these weights at inference time can induce a minor computational penalty, which can be mitigated by merging the weights post-fine-tuning. This separation can also facilitate a more flexible and efficient system, especially when hosting an LLM utilized by multiple clients for diverse applications. By maintaining separate LoRA weights for each client or application, storage efficiency is significantly enhanced, albeit with a slight compromise on performance.

With LoRA we can modify what model will do, task type´s can be following:

- **With CAUSAL_LM:** This task type could be used to develop a model that can generate VHDL code snippets based on the preceding context. It would be predicting the next word or token in a VHDL code sequence.
- **With MASKED_LM**: This task type could be used for a model that can fill in missing parts of a VHDL code snippet. You would mask certain parts of the code and have the model predict the masked parts.
- **With TOKEN_CLASSIFICATION:** This task type might be used to develop a model that can identify and classify different components of VHDL code, such as identifying keywords, operators, etc.
- **With CLASSIFICATION**: This could be used for a model that classifies VHDL code snippets into different categories based on certain criteria, such as complexity or functionality.
- **With SEQ2SEQ_LM:** This task type might be used for tasks where you want to translate VHDL code into another representation or format, such as translating VHDL code to a graphical representation or to another hardware description language.

4 IMPLEMENTATION OF THE CUSTOM LLM MODEL

In this section, we explain how the custom model was implemented, providing code examples.

As elucidated in section 3.1, training a model requires comprehensive planning. It is essential to comprehend the relationship between the model's intended use and the available data, to foster realistic and grounded expectations. Indeed, fine-tuning is a labour-intensive task, often navigated through a methodical process of trial and error.

In this thesis, we chose not to undertake human evaluations due to their time-consuming nature. Additionally, a detailed description of the deployment process is omitted since it encompasses confidential company specifics.

Given the numerous components in this work area, not all phases were necessarily executed.

The general workflow is structured as follows:

- Data Collection:
  Initiate the process by assembling a dataset that is representative of the new terms and contexts that are central to your study, with a particular emphasis on VHDL code in this instance.
- Tokenizer Training:
  Proceed to train or fine-tune the tokenizer with the gathered dataset, thereby integrating the new terms into the vocabulary, enhancing its ability to interpret and analyse VHDL code.
- Data Tokenization:
  Utilize the newly trained tokenizer to process your dataset, converting the raw text into a structured format that can be seamlessly integrated into the model, with VHDL code serving as the focal input.
- Model Fine-Tuning:

Engage in the meticulous process of fine-tuning the model, utilizing the tokenized data to adapt it to the nuances of your specific task, enhancing its predictive accuracy and reliability.

- Model Deployment:
  This phase is bypassed; we solely tested our cases with the model.

4.1 Model Architecture

4.1.1 Codegen25 architecture

The model architecture, dubbed "Codegen25", is delineated as follows:

```
LlamaForCausalLM(
  (model): LlamaModel(
    (embed_tokens): Embedding(51200, 4096, padding_idx=0)
    (layers): ModuleList(
      (0-31): 32 x LlamaDecoderLayer(
        (self_attn): LlamaAttention(
          (q_proj): Linear(in_features=4096, out_features=4096, bias=False)
          (k_proj): Linear(in_features=4096, out_features=4096, bias=False)
          (v_proj): Linear(in_features=4096, out_features=4096, bias=False)
          (o_proj): Linear(in_features=4096, out_features=4096, bias=False)
          (rotary_emb): LlamaRotaryEmbedding()
        )
        (mlp): LlamaMLP(
          (gate_proj): Linear(in_features=4096, out_features=11008, bias=False)
          (up_proj): Linear(in_features=4096, out_features=11008, bias=False)
          (down_proj): Linear(in_features=11008, out_features=4096, bias=False)
          (act_fn): SiLUActivation()
        )
        (input_layernorm): LlamaRMSNorm()
        (post_attention_layernorm): LlamaRMSNorm()
```

```
      )
    )
    (norm): LlamaRMSNorm()
  )
  (lm_head): Linear(in_features=4096, out_features=51200, bias=False)
)
```

Detailed Breakdown of components:

- LlamaForCausalLM: The main class representing the base model structure, which houses the core components of the model, including the embedding layer and the model layers.
- LlamaModel: The core model structure containing the embedding layer and various layers of the model, including the LlamaDecoderLayer and LlamaAttention components.
- Embedding(51200, 4096, padding_idx=0): The embedding layer where tokens are embedded into a higher-dimensional space, with a vocabulary size of 51200 and a 4096-dimensional embedding space(represents a space where each word in our vocabulary is mapped to a 4096-dimensional vector).
- LlamaDecoderLayer (0-31): A series of 32 individual layers within the model, each housing several components, including the self_attention mechanism and the MLP (Multi-Layer Perceptron).
- LlamaAttention: The attention mechanism in each layer, which contains several linear transformations, facilitating the focus on different parts of the input sequence when producing an output sequence.
- LlamaMLP: A feedforward neural network in each layer, which contains several linear transformations and an activation function (SiLU).
- Linear Transformations: These are the linear transformations in the attention mechanism, each performing a specific function in processing the input data.
- LlamaRotaryEmbedding: A component that facilitates the embedding of rotary positional encodings, enhancing the model's ability to capture sequential information in the data.
- LlamaRMSNorm: A normalization layer applied at various stages of the model to stabilize the activations and facilitate training.
- Linear (lm_head): The final linear layer which maps the output to the desired output size (51200, which is the vocabulary size), facilitating tasks like language modeling.

4.1.2 Fine-tuned custom model architecture.

The custom model architecture, labeled "Codegen25 with LoRA", is presented below:

```
PeftModelForCausalLM(
  (base_model): LoraModel(
    (model): PeftModelForCausalLM(
      (base_model): LoraModel(
        (model): PeftModelForCausalLM(
          (base_model): LoraModel(
            (model): LlamaForCausalLM(
              (model): LlamaModel(
                (embed_tokens): Embedding(51200, 4096, padding_idx=0)
                (layers): ModuleList(
                  (0): LlamaDecoderLayer(
                    (self_attn): LlamaAttention(
(q_proj, k_proj, v_proj, o_proj): Linear with LoRA components (incorporating lora_dropout, lora_A, lora_B, lora_embedding_A, lora_embedding_B)
                      (rotary_emb): LlamaRotaryEmbedding()
                    )
                    (mlp): LlamaMLP(
                      (gate_proj, up_proj, down_proj): Linear
                      (act_fn): SiLUActivation()
                    )
                    (input_layernorm, post_attention_layernorm): LlamaRMSNorm()
                  )
                  (1-31): LlamaDecoderLayer (without LoRA components)
                )
                (norm): LlamaRMSNorm()
              )
              (lm_head): Linear with additional components (CastOutputToFloat, 0 (Linear))
            )
          )
        )
      )
```

)
)
)

- PeftModelForCausalLM - This represents the main class post fine-tuning with LoRA, encompassing nested base models and additional components introduced by LoRA. The structure is notably nested, with several layers of base models encapsulated within each other. The primary additions are the LoRA components incorporated into the linear transformations in the attention mechanism of the first layer (layer 0). These components encompass dropout layers, supplementary linear transformations (lora_A and lora_B), and parameter dictionaries for embeddings (lora_embedding_A and lora_embedding_B).

- lm_head:
  CastOutputToFloat: A module that transitions the output data to a floating-point format, ensuring numerical stability and compatibility with subsequent layers.
  0 (Linear): The initial layer in a sequence, executing a linear transformation on its input to generate scores for each word in the vocabulary, facilitating tasks such as language modeling.

Rationale for Model Selection:
The preference for this specific architecture is rooted in its reputation as a state-of-the-art model, presenting a robust framework for language modeling endeavors. Training with VHDL has proven pivotal, enabling the model to understand and interpret VHDL code more effectively.

Causal training responds to the requisite prompting; using this modeling approach, we can generate entire code snippets. This strategic decision resonates with our goal of crafting a model not just advanced but also attuned to the intricacies and peculiarities of VHDL code analysis.

4.2 Data Collection and VHDL Code Preparation of custom model

In order to prepare our VHDL data for subsequent processing and fine-tuning, it was vital to structure the information coherently. To achieve this, we devised a script to systematically pair each VHDL code segment's description with its corresponding VHDL code.

```
data_texts = []
with open(temp_file, 'r') as input_f:
    content = input_f.read().split('###END_OF_SEGMENT###')
    for segment in content[:-1]:
        lines = segment.strip().split('\n')
        input_line = lines[0].strip()
        output_code = '\n'.join(lines[1:]).strip()
        if input_line and output_code:
            data_texts.append({
                'input': input_line,
                'output': output_code
            })
```

Procedure:

1.Reading the Temporary File: The script begins by opening the temp-file, which houses segmented VHDL data. Each segment within this file is demarcated by the separator "###END_OF_SEGMENT###".

2.Segment Parsing: For each isolated VHDL segment:

The segment is broken down into its individual lines.

The initial line, which typically describes the VHDL code's purpose or functionality, is tagged as the input-line.The subsequent lines, embodying the VHDL code itself, are collectively categorized as the output-code.

3.Data Structuring: To ensure efficient accessibility for future tasks, the parsed data is organized in a dictionary format. If both the description (input_line) and the VHDL code (output_code) are present and valid, they are added to the data_texts list as a dictionary pair.

Outcome:

The result of this procedure is a list of dictionary entries, data_texts, where each entry distinctly maps a VHDL description to its corresponding VHDL code. This structured format is not only conducive for understanding but also facilitates seamless integration into subsequent model training and evaluation phases.

Subsequently, considering our challenges concerning training time, we revised the method of data ingestion into the model. Specifically, we adjusted how sizable chunks or sequences of data are fed into the model during the training phase as this play important role how much time we spend for training.

```
selected_data = data_texts[:500000]
print("First selected data:", selected_data[0])
train_data, val_data = train_test_split(selected_data, test_size=0.2, shuffle=False)
```

Loading...

Though tokenizer training was pursued, we found its implementation redundant for the training phase. This is attributed to our realization that codegen inherently caters to code generation tasks and that VHDL, being a code language, resonated well with the base tokenizer. Consequently, the trained tokenizer was not employed in the actual training.

4.3 Fine-tuning

For the fine-tuning process, we utilized a dataset amounting to 100MB of VHDL data.
Our journey through the fine-tuning process was characterized by:

Iterative Adjustments: We underwent numerous iterations, each focusing on tweaking different LoRA parameters. A significant part of these adjustments involved adding more layers to the training process, enhancing the model's depth and complexity.

Baseload Bit Alterations: We made necessary modifications in how the model was loaded, transitioning between 32-bit to 4-bit configurations. Notably, the 32-bit version proved too voluminous to fit within the confines of our GPU's memory, necessitating these shifts.

Harnessing Data Parallelism: Given the model's initial hefty size, we leveraged data parallelism to make the most of PyTorch's GPU capabilities. This approach enabled more efficient processing and accelerated training, despite the model's complexity.

4.3.1 LoRA

Our intent to selectively train a segment of the base model parameters guided us to employ LoRA (Low-Rank Adaptation). The configuration adopted for LoRA was:

```
# lora model, all linear layers are trained
target_modules = ['q_proj','k_proj','v_proj','o_proj','gate_proj','down_proj','up_proj','lm_head']
lora_config = LoraConfig(
    r=8, #r represents the rank of the low rank matrices learned during the finetuning process. As t
    lora_alpha=32,
    target_modules = target_modules,
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM"
)
```

r= 8 : Representing the rank, it signifies the intricacy of modifications being applied to the model. A diminished rank implies fewer modifications, thereby economizing on computational resources and time.

lora_alpha=16: This is a scaling factor that helps in controlling the magnitude of the adjustments we are making. A higher value would mean more significant adjustments, potentially leading to more pronounced effects (but it might also increase the risk of overfitting).

Target_modules: Depending on the model being fine-tuned, it is essential to discern the attention layers. For our work, these were specific to the codegen25 model.

lora_dropout=0.05: A strategy to combat overfitting, it sets the dropout rate at 5%. It ensures that 5% of connections in the LoRA layers are occasionally disregarded during training, bolstering the model's generalization capability.

bias="none": This indicates that no bias is being added to the LoRA layers during the training process. Bias can sometimes help in controlling the output of the neurons, but in this case, it is not being used.

task_type="CAUSAL_LM": Since our core intent was to fashion a model proficient in comprehending and regenerating VHDL code, the task type is set to causal language modeling.

4.3.2 Model Fine-Tuning setup-phase

The fine-tuning phase is pivotal in adapting the pre-trained model to our specific dataset, enabling it to understand the nuances of our training data. Below are the training arguments with annotations:

```
# Specifies the batch size for each device during training. Bigger batch sizes allow for faster training but require more memory.
per_device_train_batch_size = 48
# This defines the number of steps to take before updating the model's weights. It's used to train with bigger effective batch sizes with limited memory.
gradient_accumulation_steps = 4
# The optimizer to use. 'adamw_hf' is a variant of the Adam optimizer with weight decay fix, commonly used in transformer models.
optim = 'adamw_hf'
# The learning rate for the optimizer. It determines the step size at each iteration while moving towards a minimum of the loss function.
learning_rate = 3e-5
# The maximum value for gradient clipping. Helps in preventing the "exploding gradients" problem.
max_grad_norm = 1
# The proportion of the total training time where the learning rate increases (warmup period). After this period, the learning rate starts to decay.
warmup_ratio = 0.03
# Specifies the learning rate scheduler type. A linear scheduler decreases the learning rate linearly.
lr_scheduler_type = "linear"
# Weight decay is a regularization technique. It adds a small penalty on the weights magnitude, preventing them from growing too large.
weight_decay=0.01
# Indicates the device type to use for training. "cuda" implies using NVIDIA GPUs for training.
device = "cuda"
# Moving the model to the specified device.
model = model.to(device)
# Arguments for the training process.
training_args = TrainingArguments (
    output_dir=base_dir,  # Directory where training outputs will be saved.
    logging_dir=logging_dir,  # Directory for logging training metrics and statistics.
    eval_steps=50,  # Number of steps before an evaluation is performed. Originally set to 10.
    save_steps=50,  # Number of steps before the model checkpoint is saved.
    num_train_epochs = 30,  # Total number of training epochs.
    per_device_train_batch_size=per_device_train_batch_size,
    gradient_accumulation_steps=gradient_accumulation_steps,
    optim=optim,
    weight_decay=weight_decay,
    learning_rate=learning_rate,
    max_grad_norm=max_grad_norm,
    group_by_length=True,  # Groups sequences of similar lengths together for more efficient padding.
    evaluation_strategy="steps",  # Evaluation strategy set to evaluate at specified step intervals.
    save_strategy="steps",  # Save strategy set to save model at specified step intervals.
    lr_scheduler_type=lr_scheduler_type,
    save_total_limit=1,  # The maximum number of checkpoints that can be saved.
    fp16=True,  # Enables half-precision training for faster performance (requires NVIDIA Apex library).
    #load_best_model_at_end=True,  # If uncommented, will load the best model at the end of training.
)
```

Loading...

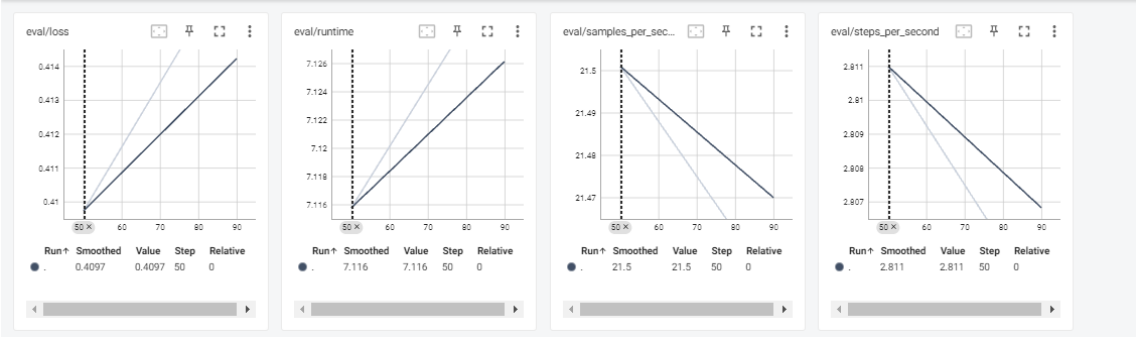
Trainer setup:

```
# Prepare the model for distributed and mixed precision training with Accelerate.
# Accelerate is a library that helps in simplifying GPU/CPU distributed training.
model = accelerator.prepare(model)
# Initialize the Trainer class from Hugging Face's Transformers library.
# This class provides functionalities for training and evaluating transformer models.
trainer = Trainer(
    model=model,                # The model to train.
    args=training_args,         # Training arguments defined earlier.
    train_dataset=train_dataset, # Training dataset.
    eval_dataset=val_dataset,    # Validation dataset.
    data_collator=data_collator, # Handles batching of data.
    tokenizer=tokenizer,        # Tokenizer for encoding the data.
)
# Upcast layer normalization to float 32 for stability during training.
# Without this, training can sometimes produce NaN values due to float16 precision.
for name, module in trainer.model.named_modules():
  if "norm" in name:
    module = module.to(torch.float32)
# Start the training process.
trainer.train()
# Evaluate the model on the validation dataset.
results = trainer.evaluate()
# Clear GPU memory cache to free up resources.
torch.cuda.empty_cache()
# Print the evaluation results.
print(results)
```

As previously mentioned, we executed training multiple times due to the myriad of variable parameters. A significant challenge was the extended training duration. Often, projections indicated that training would span several weeks. Ultimately, we identified that the test data was the bottleneck – it was "raw" and hadn't been sufficiently processed or labelled for efficient integration into the model.

We kept a vigilant eye on the training process using a series of metrics, crucial for evaluating the model's learning trajectory. These metrics illuminated the model's learning efficiency and indicated if tweaks were essential to avoid pitfalls like overfitting or underfitting. The loss function's visualization, across training epochs, was facilitated using Tensorboard and MLRUNs, both supported by the Huggingface library. Additionally, we implemented custom logging to monitor GPU memory usage during training.

Given that our code execution happened within containers on a Unix environment, real-time monitoring of training progress was limited. Thus, tools like Tensorboard and custom logging were invaluable. For one training iteration, the loss was documented as 0.4172954559326172. An illustrative snapshot of the Tensorboard metrics is provided below (it is crucial to note that our validation steps were set at 50, with overall steps at 90, which explains the unconventional logging pattern).



4.3.3 Logging during training

Here is sample Logs from the Training Phase

GPU memory usage:

2023-10-30 07:29:08 - Memory usage on GPU 0: 12350.05 MB

2023-10-30 07:39:08 - Memory usage on GPU 0: 13884.01 MB

2023-10-30 07:49:08 - Memory usage on GPU 0: 9848.76 MB

2023-10-30 07:59:08 - Memory usage on GPU 0: 13072.25 MB

Model Architecture

base_model.model.model.norm.weight: mean=1.248884677886963, std=0.09756681323051453

base_model.model.lm_head.weight: mean=-5.098325709695928e-05, std=0.02528771571815014

base_model.model.lm_head.lora_A.default.weight: mean=5.730737029807642e-05, std=0.009078376926481724

base_model.model.lm_head.lora_B.default.weight: mean=-4.071862349519506e-05, std=0.00041720899753272533

Trainable Parameters Statistics:

2023-10-30 06:40:45 - After we have enambed k_bit training: trainable params: 0 || all params: 6895702016 || trainable%: 0.0

2023-10-30 06:41:28 - After LoRA modification: trainable params: 20430848 || all params: 6916132864 || trainable%: 0.29540855275275407

2023-10-30 08:05:29 - After Training: trainable params: 20430848 || all params: 6916132864 || trainable%: 0.29540855275275407

Upon completing the training, our next task was to integrate the base model with an adapter. For readers unfamiliar with the concept, an adapter is a compact layer introduced between the pre-existing layers of a pre-trained model. Instead of extensively fine-tuning the entirety of the pre-trained model, we only adjust the adapter's parameters. This method capitalizes on the advantages of the pre-trained model while providing flexibility to adapt to new tasks, all with minimized computational overhead. Before this integration, we ensured the model was securely saved.

Subsequently, we embarked on the "inference" phase, colloquially understood as model testing.

4.3.4 Conclusion and discussions

Fine-tuning our model proved to be a complex and iterative endeavor. The journey required a substantial time investment, with the entire training process spanning roughly two months. This extensive duration underscores the commitment and vast computational resources that were dedicated to achieving our goals.

Operating in the LSF and Docker environments presented unique challenges. In comparison, platforms like Jupyter notebooks typically offer more immediate feedback and accessibility to results. As we embarked on this path, we encountered memory-related challenges when attempting to deploy our model on dual GPUs. This prompted us to adopt a single GPU setup, leading us to further adjust the base model. By converting it to load 8-bit parameters onto the GPU, we successfully optimized memory consumption. To put it in perspective, this 8-bit model consumed only 8GB, a stark contrast to the whopping 26GB required by its 32-bit counterpart.

Through several trials and adjustments, especially concerning the batch size, we managed to whittle down the training duration to a more manageable few day. Several insights emerged from this coding phase. For instance, the Huggingface Trainer, while robust, contains several parameters that aren't always directly visible. However, with some investigation, these can be identified within the 'mlruns' folder.

We recognized early on the value of custom logging, primarily because we frequently grappled with "out of memory" issues. This phase also underscored the importance of vigilance; it is crucial to continuously verify that the GPU is actively being utilized. Although physically present, it is not always guaranteed that the GPU is in operation. Additionally, the utilities offered by the Hugging-face Trainer class, such as model-saving based on 'eval_steps', proved invaluable, especially when monitoring metrics that eventually plateau.

One of the foundational insights from this process was the absolute importance of understanding our data. Comprehensive knowledge of the data's nature and structure informed various decisions, from categorization and labelling to other essential preprocessing measures. We also delved deep into optimizing memory during the training phase, exploring various methods and parameters to enhance performance and resource utilization.

Upon concluding the training, we observed the surprisingly compact size of our 'adapter.bin' model, which was approximately 80MB. This initially unexpected size was rationalized by the realization that our model had a relatively modest count of 20,430,848 trainable parameters. To integrate the subtle yet crucial changes from our fine-tuning process, we found it imperative to merge the base model with the adapter.

Reflecting on this phase, it was a period filled with challenges, insights, and continuous learning. Over two months, we engaged with multiple code iterations, refining our approach and strategies until we achieved our desired outcomes.

5 TESTING OF MODELS

In the realm of Machine Learning, "testing" is often referred to as "inference." It entails feeding test cases into the models and recording the output in a designated Excel file that houses all test cases and corresponding results. Expert reviews then evaluate these outputs to determine if the models are delivering expected results. The decision is binary – either true (correct) or false (incorrect). This evaluation does not allow for back-and-forth discussions; it is a direct assessment based on the model's output.

The codegen25 supports two sampling methods: Causal and Infill. Given our objective to produce code, we opted for causal sampling. In contrast, infill sampling would be more apt for tasks like commenting on code or injecting snippets into pre-existing code.

5.1 Description of Test Cases

We formulated a set of 10 cases, divided across different levels of VHDL complexity: four basic VHDL tasks, four intermediate tasks, and two advanced tasks.

The decision-making process was strictly quantitative. If a model's output was 99% accurate, it was scored a '1'. Anything less resulted in a '0'.
The test cases encompassed tasks like:

```
texts = [
    -- create a VHDL code of a single wire
    -- create a VHDL code of a 2-input and gate,
    -- create a VHDL code of a 3-bit priority encoder,
    -- create a VHDL code of a 2-input multiplexer,
    -- create a VHDL code of a half adder,
    -- create a VHDL code of a 1-to-12 counter,
    -- create a VHDL code of a LFSR with taps at 3 and 5,
    -- create a VHDL code of a FSM with two states,
    -- create a VHDL code of a Signed 8-bit adder with overflow,
    -- create a VHDL code of a Counter with enable signal,
]
```

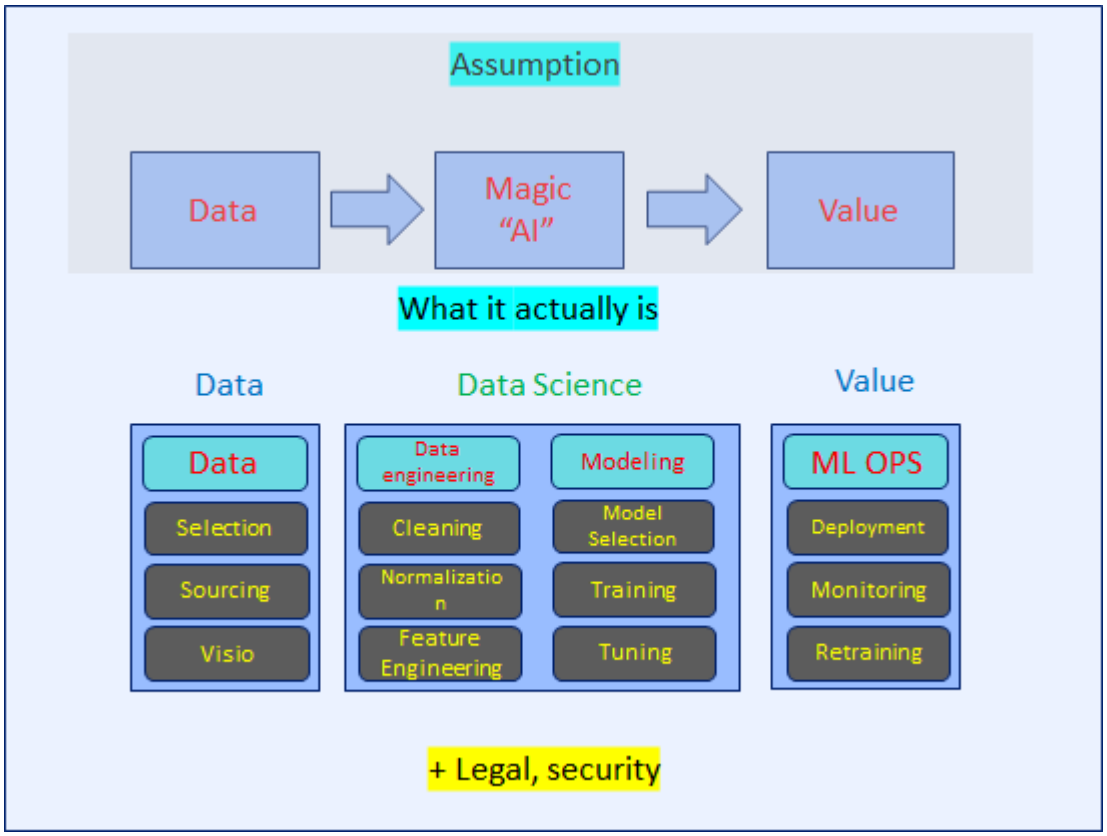
5.2 Results

Problem #	Difficulty	Description		Internall LLM	Base Model(Codegen)	ChatGTP4
1	Basic	A simple wire		0	0	1
2	Basic	A 2-input and gate		0	0	1
3	Basic	A 3-bit priority encoder		1	0	1
4	Basic	A 2-input multiplexer		1	0	1
5	Intermediate	A half adder		0	0	1
6	Intermediate	A 1-to-12 counter		0	0	1
7	Intermediate	LFSR with taps at 3 and 5		0	0	0
8	Intermediate	FSM with two states		0	0	1
9	Advanced	Signed 8-bit adder with overflow		0	0	1
10	Advanced	Counter with enable signal		0	1	1
			Sum	2	1	9

From our analysis, it is evident that ChatGPT-4 surpasses all other contenders, delivering unparalleled performance. Given our strict criterion – code either works perfectly or it does not – there is no room for approximations. Several models came close, generating code with minor bugs or truncated outputs due to prompt limitations. However, for a fair comparison, we maintained consistent prompting across all models.

6 CONCLUSION

Our initial outlook at the onset of this journey can be best captured by the illustration below. It paints a vivid picture of our naive optimism: It was initially presumed that providing the model with a substantial volume of raw VHDL data would enable it to learn seamlessly.



Contrary to these initial expectations, the modelling process was complex. While our original inclination was to believe that the more data we threw at the model, the better it would perform, this strategy proved to be more apt for creating models from scratch rather than for the intricate process of fine-tuning.

Initially, the strategy was to inundate the model with a plethora of VHDL data, anticipating that it would adapt and "learn." However, the reality pointed towards the approach of unsupervised learning being more suitable for constructing a model from scratch rather than fine-tuning an adapter for a specific outcome. Consequently, the significance of supervised learning emerged, which in essence, meant training with labelled data, pairing each input question with the desired VHDL code output.

Creating the internal model proved to be a challenging endeavour. The task of maintaining its accuracy, particularly when operating within a UNIX environment coupled with a Docker container, was significantly more demanding than working on a personal computer equipped with tools such as Visual Studio and Jupyter Notebook (where results are immediately observable). This extensive fine-tuning journey lasted approximately two months. Due to the intricate nature of VHDL, our primary objective shifted towards ensuring the model's functionality, even if that meant occasionally limiting ourselves to simpler test cases.

Tokenization was among the standout challenges. Despite numerous trial-and-error attempts at training the tokenizer, it was eventually omitted from the process. This decision stemmed from recognizing that VHDL, akin to C-code, didn't necessitate specialized tokens. Additionally, sourcing an adequate dataset while maintaining stringent data security presented its own set of challenges.

The testing phase, focused on a select set of basic VHDL cases, didn't yield the optimistic results we had anticipated. However, it established a foundation for future endeavours. The experience underscored the importance of data labelling, which hastened the training process but demanded multiple iterations.

In an ideal world, a multidisciplinary team of ML engineers, data engineers, and other specialists would be deployed to undertake such a project, equipped with a clear vision of the data at hand and the desired project outcomes.

Key Takeaways:

- **Consistently log work**. Notably, the Huggingface trainer generates logs even without any supplemental coding.
- **Regularly** monitor model weights pre and post-training.
- **Ensure to log the inference** (testing) process to guarantee the correct loading of both the trained adapter and base model.
- **Confirm that computations** are GPU-accelerated, using specific commands like “model.to.gpu”.

- **The quantity and treatment** of input data profoundly affect the training duration. It is pragmatic to start with smaller datasets and fewer epochs to gauge the direction. This was our most significant learning.
- **Gradually enhance** the data input for improved model generalization.

Future work:

- **Use the SSTF-trainer** designed specifically for training supervised (or labeled) data. This implies that the incoming data should be more meticulously "engineered", potentially aligning with the Alpaca mode which incorporates instructions, input, and output, using these labels as a guide for teaching.

- **Experiment with different base models**, including those not inherently tailored for coding, to evaluate their performance and outputs.

- **Devote more attention to data labeling**. This was identified as a significant bottleneck since we began with a voluminous dataset and an excessive number of epochs.

- **Engage in quantization experiments.** For instance, assess the impact of training with 16-bit versus 32-bit precision. Likewise, investigate the implications of loading the base model in 4-bit and other configurations. Preliminary findings suggest that such quantization at the very least makes it feasible to train the model on a single NVIDIA 40G GPU.

7 REFERENCES

all, A. V. (2023). Attention is all you need. 15.

all, E. H. (2021). LORA: LOW-RANK ADAPTATION OF LARGE LANGUAGE MODELS. *LORA: LOW-RANK ADAPTATION OF LARGE LANGUAGE MODELS*, 26.

all, E. N. (2023). CODEGEN2: LESSONS FOR TRAINING LLMS ON PROGRAMMING AND NATURAL LANGUAGES.

all, S. T. (2022). Benchmarking Large Language Models for. *Automated Verilog RTL Code Generation*, 7.

contact@bigcode-project.org. (2023). S TA R C O D E R : may the source be with you. 54.

Open AI. (2023). GPT-4 Technical Report. 100.

APPENDIX

A. VHDL Code Samples

B. Python code structure

APPENDIX A

```
-- VHDL File: example.vhdl
-- Author: Your Name
-- Description: This file contains a mix of VHDL code, comments, errors, and documentation.

-- LIBRARY DECLARATION
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL:

-- ENTITY DECLARATION
entity Main is
    Port (
        clk    : in STD_LOGIC;
        reset  : in STD_LOGIC;
        input  : in UNSIGNED (7 downto 0);
        output : out UNSIGNED (7 downto 0)
    );
end Main;

-- ARCHITECTURE DEFINITION
architecture Behavioral of Main is
begin
    process(clk, reset)
    begin
        if reset = '1' then
            output <= (others => '0');
        elsif rising_edge(clk) then
            output <= input; -- Direct assignment is fine since both are UNSIGNED now
        end if;
    end process;
end Behavioral;
```

-- DOCUMENTATION

-- In VHDL, the entity declaration is used to declare the external interface of a design unit, including the name of the design unit, the names, modes, and types of its ports.

-- ERROR MESSAGES

-- ERROR: [VRFC 10-91] 'std_logic_arith' is not compiled in library IEEE

-- VHDL TIPS

-- 1. Comments in VHDL start with two hyphens (--).

-- 2. The library declaration section is used to include libraries that contain packages to be used in the design.

-- 3. The entity declaration defines the interface of the VHDL module.

-- 4. The architecture body contains the actual implementation of the entity.

-- END OF FILE

APPENDIX B

Given that the specific code developed during this study is confidential, this section will only provide a general overview of the code's structure:

1. Imports: Importing necessary libraries and dependencies.
2. Loading the Base Model: Initializing and setting up the foundational model.
3. Creating the LoRA Configuration: Setting up the configuration for LoRA.
4. Vocabulary Retrieval: Obtaining the vocabulary of the base model. (Note: This step was ultimately skipped as it became optional.)
5. Tokenizer Training: Training the tokenizer with specific keywords. (Note: This step was also deemed optional and skipped towards the end.)
6. Data Modification: Adjusting and preparing the training data for the model.
7. Data Ingestion: Inputting the training data and splitting it into training, testing, and (if necessary) validation subsets.
8. Defining Training Arguments: Specifying parameters and arguments for training.
9. Model Training/Fine-Tuning: Initiating the process to train or fine-tune the model or adapter.
10. Results Evaluation: Analyzing and assessing the outcomes post-training.
11. Model Saving: Storing the trained model for future use.
12. End User Testing: Conducting inference tests to see how the model performs in real-world scenarios.