BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

GENERATING CODE FROM TEXTUAL DESCRIPTION OF FUNCTIONALITY

GENEROVANÍ KÓDU Z TEXTOVÉHO POPISU FUNKCIONALITY

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. JAN ŠAMÁNEK**
AUTOR PRÁCE

SUPERVISOR **Doc. RNDr. PAVEL SMRŽ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

146415

Institut:	Department of Computer Graphics and Multimedia (UPGM)
Student:	**Šamánek Jan, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Machine Learning
Title:	**Generating Code from Textual Description of Functionality**
Category:	Information Systems
Academic year:	2022/23

Assignment:

Literature:

- according to the supervisor's recommendations

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Smrž Pavel, doc. RNDr., Ph.D.**
Head of Department:	Černocký Jan, prof. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	17.5.2023
Approval date:	31.10.2022

Abstract

As machine learning and neural network models continue to grow, there is an increasing demand for GPU-accelerated resources and algorithms to support them. Large language models have the potential to assist with this task, as they are already used as coding assistants for popular programming languages. If these models could also learn less commonly used paradigms like CUDA, they could help develop and maintain the necessary systems. This thesis aims to explore the capabilities of modern language models for learning CUDA as a programming paradigm and creating a training corpus specifically for this purpose.

Abstrakt

S pokračujícím nástupem strojového učení a stále větších modelů neuronových sítí, roste i potřeba GPU akcelerovaných zdrojů a algoritmů pro podporu těchto modelů. Vzhledem k tomu, že velké jazykové modely jsou již dnes využívány jako asistenti při programování v moderních programovacích jazycích, mohli by s tímto problémem pomoci. Pokud by se tyto modely dokáží naučit i méně známá paradigmata, jako je CUDA, mohly by pomoci s vývojem a udržování těchto systémů. Tato práce zkoumá schopnosti moderních jazykových modelů pro učení se CUDA jako programovacího paradigmatu a také vytvoření nové trenovací sady, určené pro tyto účely.

Keywords

Machine learning, NLP, CUDA, GPU, Transformer, Code generation, Dataset

Klíčová slova

Strojové učení, NLP, CUDA, GPU, Transformer, Generování kódu, Dataset

Reference

ŠAMÁNEK, Jan. *Generating Code from Textual Description of Functionality.* Brno, 2023. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Doc. RNDr. Pavel Smrž, Ph.D.

Generating Code from Textual Description of Functionality

Declaration

I hereby declare that this thesis was prepared as an original work by the author under the supervision of associate professor Pavel Smrž. The supplementary information was provided by associate professor Jiří Jaroš. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....

Jan Šamánek

May 13, 2023

Contents

List of Figures

Chapter 1

Introduction

In this thesis, we will look at code generation from the textual description of the functionality, more precisely, at generating the code for the graphical cards in CUDA language. The motivation of this research is to create and analyze a corpus focusing on the CUDA kernels that can be used for finetuning the AI systems for the CUDA-generating task and evaluating its performance by using it for finetuning several selected models of different sizes and origins. The second motivation of this thesis is to also look at the modern language models' (LMs) capabilities of learning unusual and unorthodox paradigms in coding that differ from the more mainstream languages such as Python or Java.

At the beginning of the thesis, we familiarize the reader with the necessary basics of Natural Language Processing (NLP), machine learning, and the process of creating new datasets. We describe the tools used in theory, such as the used neural network architecture (transformer), the tokenization process, and the used metrics for evaluating our system, including one newly created, to indicate the generated code's syntax correctness. We also discuss the CUDA language as a programming paradigm and explain the difference between CUDA and other languages. We will discuss the key features of CUDA and crucial aspects that the programmer should know about the language in order to write efficient and bug-free code.

A significant part of this thesis is dedicated to the process of creating the corpus, as the dataset is the backbone of machine learning, and its quality is essential for creating a well-performing system for any downstream task. We discuss the process of obtaining and processing data as well as the extensive analyses of the data and its source quality. After that, we propose the system for training and evaluating the selected models and explain the pipeline. We cross-validate our corpus usability with several selected modern neural networks of different sizes, architectures, and backbone corpora. We introduce the reader to each of the models, briefly explain the model's origins, and discuss the results of the training & validation process.

After evaluating the selected models, we analyze some of the errors and try to summarize the models' performance. We show the performance samples for each model in the appendix, including the final comparison of the best-performing model with the current state-of-the-art large language model GPT4. In conclusion, we briefly summarize the extent to which expectations were met and offer possibilities for further research and improvement of the results so far.

Chapter 2

Natural Language Processing

In the first part of this thesis, it would be wise to talk about the science that deals with the code generation problem. The book „Fundamentals of Artificial Intelligence“ by K. R. Chowdhary [12] defines Natural Language Processing, or NLP, as:

> Collection of computational techniques for automatic analysis and representation of human languages, motivated by theory.

Even though it paints a picture, this definition could be more explicit, and it is hard to imagine what it means. From a more practical point of view, NLP is a science that tries to achieve a deeper understanding of human language and teach this understanding to computers. Tasks that fall into this category are, for example:

- Machine translation from one language to another,
- Answering questions,
- Spam detection,
- Sentiment analysis,
- and much more

Most of the tasks in NLP are essential for humans, and we do them daily, but as the reader can imagine, these tasks are severely complex to solve for computers. It is almost impossible to describe these tasks analytically, so we can define an algorithm to solve them with acceptable accuracy. That means today's algorithms used for NLP are leveraged mainly through machine learning.

2.1 Machine Learning

Today's machine learning algorithms dominate the NLP field the same way as almost any other part of the IT industry. Machine learning shifts software development more toward how people learn; therefore, it is more natural and more accessible to solve and generalize complex problems than by using analytical algorithms.

In the case of NLP, the most used machine learning algorithm nowadays are neural networks. These discriminative models are widely used throughout the science field for

all sorts of tasks. In this thesis, we primarily use neural network architecture called a transformer, described later in its own section 2.6.

However, neural networks also bring some drawbacks to the table. Data analysis and preparation are crucial parts of the science of neural network training. We often need big chunks of structuralized, sensible data covering the most considerable amount of the task use cases possible. This is because the neural network, during the training process, tends to search for the easiest solution, which often results in getting stuck in a local minimum of the optimization function or, in the case of a small dataset, memorizing the entire training set instead of adequately understanding the task. There are, of course, methods to deal with this problem, such as regularization techniques. However, ultimately the neural model can only be as good as the data provided in the training process.

2.2 Neural Networks

In this section, we briefly introduce the reader to the world of Neural networks, but it should be noted that this introduction is very light, and the reader should use it as a motivation to research this topic further.

Using the book by Kevin Gurney, „An Introduction to Neural Networks“ [16] as a reference, we can define a neural network as the topology of simple processing elements (nodes) whose functionality is loosely based on the neurons in the brain of humans and animals. This topology of artificial neurons then implements a complex mathematical function that transforms the input mathematical tensors into a desired output.

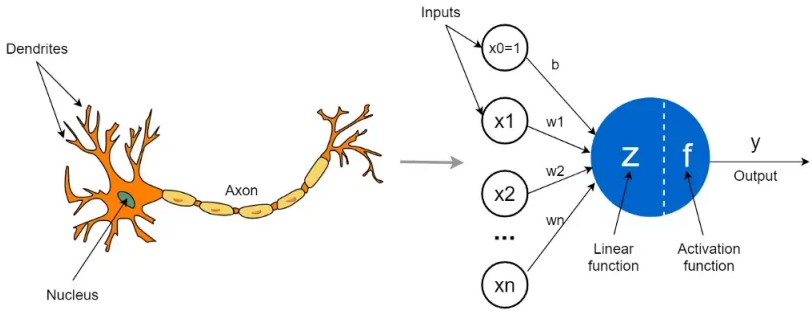


Figure 2.1: Representation of the biological neuron (left-side) and its simplification used in neural networks (right side) [35].

It is crucial to understand how artificial neurons, or as often called perceptrons, work. As the model for better visualization, we will use the right side of the figure 2.1. The perceptron is a simple math function that takes n input values and gives out one singular value. The image of the input values is implemented as the weighted sum of the input values plus bias and transformed using a (non-)linear function of choice as demonstrated in the equation below 2.2.

The equation above represents the basic building block of neural networks. The bias and the weights of the perceptron are fitted during the training phase of the network using **G**radient descent and **B**ack-propagation algorithms. To understand how we can use these algorithms to fit our neural network models, we need to define one more concept: the Loss function.

$$y = \delta(\sum_{i=0}^{n} w_i x_i + b)$$

y	Output of perceptron,
n	Number of input values,
w	Weight,
x	Input value,
b	Bias,
δ	(Non-)linear projection (e.g. sigmoid).

Figure 2.2: Mathematical demonstration of perception functionality [16].

When it comes to correcting the output of a network, we can think of a set of functions that can help us do so. These functions are known as loss functions. The loss functions indicate how far the network output is from the ideal or ground truth output. If the network output is close to the optimal solution, the loss value will be low, but if there is a significant difference, the loss value will be high. The most crucial aspect of loss functions is that they must be continuous and derivable across the whole definition range.

Once we compare the optimal result and the output from our neural network, we can calculate the loss function. Based on these loss values, we can then calculate gradients and use the Back-propagation algorithm to propagate them from the end to the start of the network. This is where we adjust the weights and biases of the network by the calculated gradients, a process known as Gradient Descent. The goal is to find the minimum of the loss function for all the training data by stepping down on the function surface. The entire process is illustrated in the accompanying figure 2.3 below.

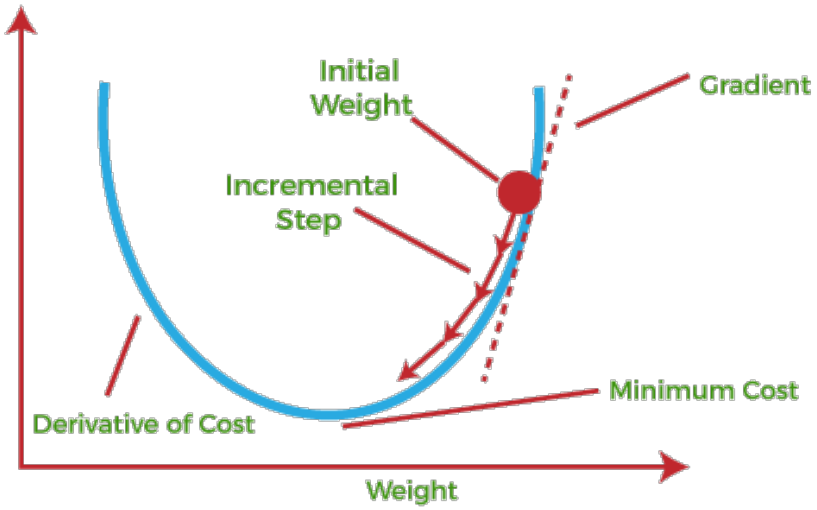


Figure 2.3: Visualization of simple gradient descent [2]. The initial weights are usually chosen randomly, and the incremented step size can be adjusted during the training process based on certain optimization techniques[1].

[1]Various Optimization Algorithms For Training Neural Network.

2.3 Language Models

Non-surprisingly, language models are particular types of architectures used in NLP. Emphasizes models plural because it is still a collection of different neural but also non-neural-like architectures. Using a more formal definition [20], language models are statistical models that calculate the validity/relevance of the text parts given as input to the model. The validity, in this case, does not necessarily mean grammatical correctness but rather the value of interest to the task the model has been trained on. Language models' advantage and the key feature is that they can theoretically analyze a text of any length.

Older language models can generate output sequences; they were/are used primarily for classification and regression tasks using information retrieval from the input text. We can classify these models into so-called N-gram models [39]:

- Unigram - $P(w_{1...n}) = P(w_1)P(w_2)...P(w_n)$
- Bigram - $P(w_{1...n}) = P(w_1)P(w_2|w_1)P(w_3|w_2)...P(w_n|w_{n-1})$
- Trigram - $P(w_{1...n}) = P(w_1)P(w_2|w_1)P(w_3|w_2, w_1)...P(w_n|w_{n-1,n-2})$

The equations above show a high-level look at how the statistical language model calculates its output sequence. The Unigram model assumes that every word is independent and does not take a count of the order of words. This means we would get the same output for any order of the same text parts. The Bigram model does not make this assumption, but they look only to look at the immediate neighbor.

Newer, generative models are advanced in this topic. For example, they can work with input information more efficiently because of the Attention layers described in the section about Transformers 2.6. As an example of some famous language model architecture, we can list Markov chains, Recurrent neural networks, or Transformers, which are used in this thesis and described more in the section 2.6.

This report will primarily describe how the language models work in sequence-to-sequence tasks, as it is the backbone behind code generation.

2.4 Tokenization

Before we dive into the transformers, we should first discuss the inputs of these models. This section is closely related to the section on word embedding. According to the study from BigScience research group [30], the tokenization process splits the input sentence into atomic words/tokens. Words in this context do not necessarily mean literal words from a language but rather subwords created from parts of the text.

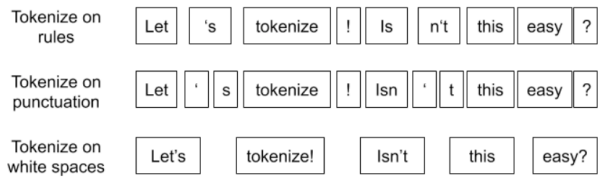


Figure 2.4: Example of tokenization of input sentence [22].

According to the same study [30] until recently, the tokenization of input sentences has been done per word. Nevertheless, this solution is not ideal from the perspective of a

significant number of words in a language; another problem is, for example, slang, newly created words, or punctuation which the model would not know how to interpret, i.g. out of vocabulary words. That is where the tokenization process comes to play. The research by Taku Kudo [24] explains different types of encoding used in tokenization. Here are some of the most common:

1. **White-space encoding**: The first and oldest way is white-space tokenization which is the easiest and represented in the figure above 2.4. White-space encoding splits the document into tokens on a word basis.

2. **Byte-pair encoding (BPE)**: The second example is **Byte-pair encoding** tokenization, which splits the document accordingly to the most often occurring N-grams in the document. Contrary to white-space encoding, Byte-pair encoding must be trained on the corpus to learn the occurrences.

3. **Unigram encoding**: The Unigram model also has to be trained the same as BPE, but the method works opposite to the BPE. The BPE model merges the most probable/occurring pairs until it reaches the desired vocabulary size. The Unigram starts with the maximum size vocabulary and excludes the least probable pairs until it reaches the desired vocabulary size.

4. **Word piece encoding**: The word piece encoding is very similar to the BPE but differs in the way of choosing the pair to put into a vocabulary. WordPiece chooses a symbol pair at each iterative step that will result in the most significant increase in likelihood upon merging. Maximizing the possibility of the training data is equivalent to finding the symbol pair whose probability divided by the probability of the first, followed by the probability of the second symbol in the pair, is more significant than any other symbol pair.

The tokenizers are usually trained on the source corpora on which we want to train our models. This technique will ensure that we would not have to deal with „Out of vocabulary“ words that the tokenizer does not know how to encode.

Another essential thing to mention about tokenizers is extending the tokenizer vocabulary with defined special tokens. This extension can be beneficial because we can create meta-data tokens for the beginning or end of our sentences, we can define padding tokens so that the input sentences in the batch to our model have the same size, or we can predefine some of the common tokens, for example, in the programming language to improve tokenization performance.

2.5 Word Embedding

Before discussing the methods and models used for the NLP tasks in greater detail, we should first describe how we represent the input sentences and documents as the data and why.

Natural language models usually cannot accept text as input. We need to find a way to transform the input document into numbers, more precisely, vectors and tensors. For this purpose, we use the method called word2vec.

According to a study by Thomas Mikolov [31], the method of word embedding is the method of transformation of the word to an n-dimensional vector space. This vector is

usually randomly selected when initialized and then finetuned during the training process. Mikolov invented neural network architecture for this purpose, which he called Word2Vec.

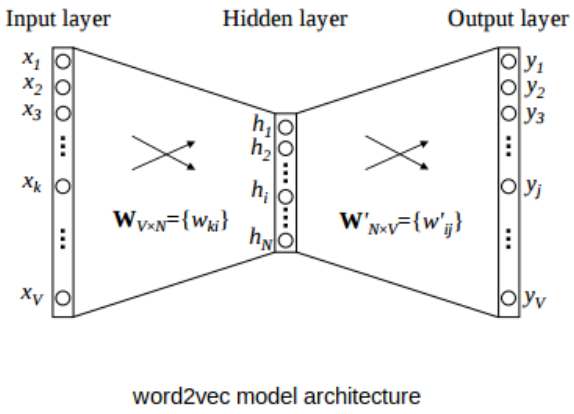


Figure 2.5: Word2vec model architecture [31].

The reason for word embedding is to map words of similar meaning closer to each other in the latent vector space, which creates a hidden algebraic structure where we can perform mathematical operations. This can be represented by the famous equation taken from [4]:

$$\text{king} - \text{man} + \text{woman} = \text{queen}$$

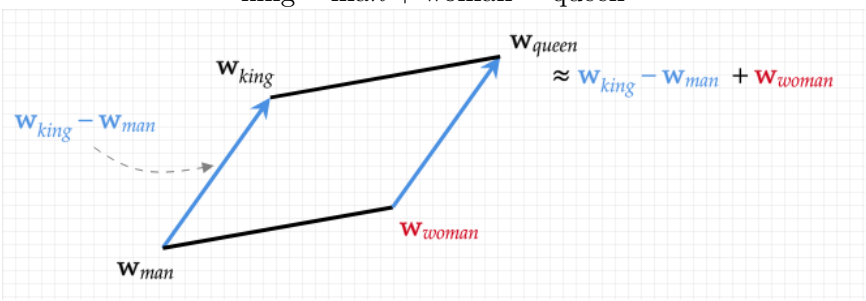


Figure 2.6: Representation of words in vector space [4].

There are several main methods for achieving this grouped mapping. Arguably the most famous ones are Bag of Words ([C]BOW) and (Continuous) Skip-gram. These methods differ in how the Word2vec model is trained.

The Bag of Words method consists of inputting N one-hot vectors, which represent the neighbor words of our target word in the sentence to the Word2vec model, and the target of the model is one one-hot vector which represents the target word.

The Skip-gram method works the opposite way. The input to the model is one word, and the target output is N neighboring words.

Besides these two methods, we can also mention fastText and Global Vectors for Word Representation (GloVe).

The previously mentioned embedding methods have one thing in common. They all are context-independent. That means that the encoding for the given token is non-variable depending on the neighboring tokens. We should change that as the given word can, for example, have different meanings depending on its context.

One way to achieve context-aware embedding would be by using convolution layers. The convolution layers are mentioned in this study mainly to name more than one method for creating context-aware embedding and are not described further in the text.

The other way to achieve context-aware embedding is by using self-attention. This method is more complex than the application of convolution filters and is further described in subsection 2.6.1.

2.6 Transformers

In this section, we finally have a look at the Transformer neural network architecture as a successor to the Recurrent neural networks [7], which were considered to be the best language models for a long time, but all changed in 2017 with the publishing of the famous paper „Attention is all you need“ [41]. In this paper, the authors proposed new architecture called “Transformer” which uses a new NN “Attention” layer. The idea behind this paper and the Attention layers is that human attention to input information is not uniform throughout all the data but rather more focused on more important details and less on other less important details. Of course, choosing which data is vital for the performed task needs to be trained. That is where the Attention layers come to play.

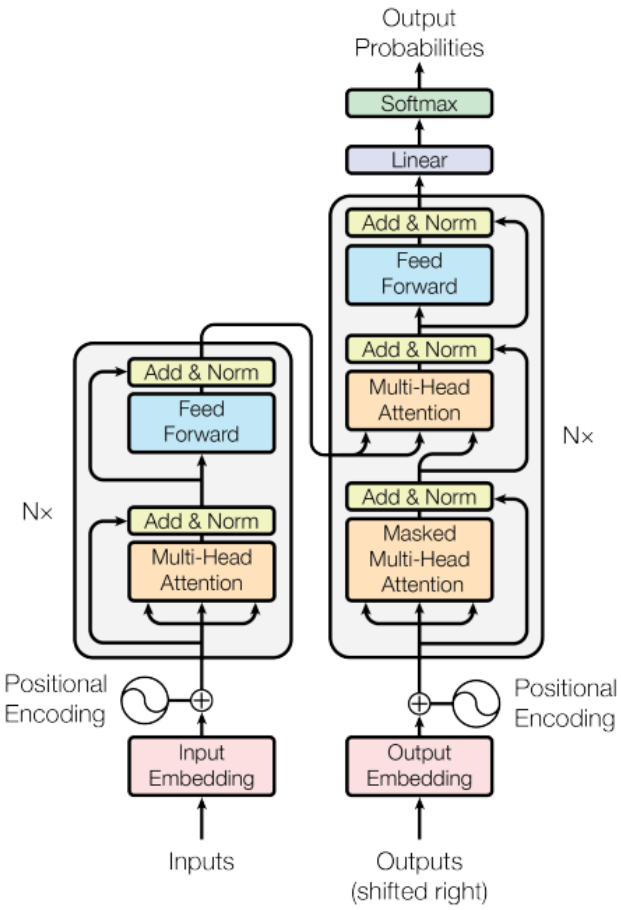


Figure 2.7: Diagram of the Transformer architecture [41].

The language models, such as the transformer and the RNN, can work in 2 modes during training. The first, **auto-regressive** mode, is when we do not know the correct output sentence, and we use the output word of the transformer as the input for the word next. The second mode is used mainly during training when we know the ground truth value. Fitting the network using this mode is called **Teacher forcing**, and instead of using the output of the transformer as the input, we use the next word of the ground truth sentence. This process can be helpful in the quicker fitting process of the network but also faster inference as we can evaluate all the input words in parallel.

There are two main parts that formes the transformer architecture. The Encoder and the Decoder 2.8, Although the model can still be used only with one of these two models.

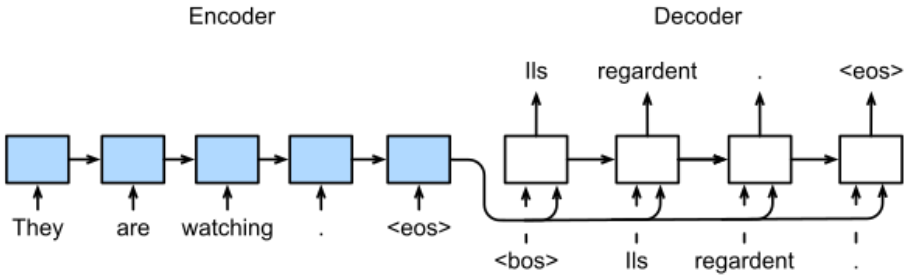


Figure 2.8: Example of auto-regressive seq2seq model [1]. The Encoder creates the representation of an input, and the Decoder, based on this representation, creates the corresponding output.

The Encoder part transforms the input sentence into the latent space where each word/-token of the sentence is represented as a vector of feature values. This output is called the Input embedding. The encoder uses self-attention layers to figure out which parts of the input prompt are the most important for the performed task.

The Decoder then takes the Input embedding (Encoder output) and uses it as a context for generating an output sequence. The sequence is decoded token by token, and for each new token, we get attention values for every input token represented as the input embedding. This creates, in the end, the so-called attention matrix. This matrix is shown in figure 2.9 for better visualization of how the transformer „thinks“ about the input data.

2.6.1 Attention

We used a lot of the “attention” expression in this study so far. In this section, we finally address this mechanism and describe what it does and how precisely it does it in great detail. This part of the study refers to the same paper, „Attention is all you need“ [41] as the previous section.

Before describing different kinds of attention, we should formally define what attention function in NLP is. The attention is defined by the „Attention is all you need“ paper [41] as:

Even though this equation tells us precisely what the mechanism is and how it works, Let us transform this equation into something more illustrative for better understanding 2.11.

Figure 2.11 shows the attention function performed on each word in the Query and for all words in Keys. First, we calculate the dot-product for every pair of tokens between Query and Keys. This calculation gives us the first glance of token importance for every token from the Query. Then we divide by the square root of the dimension of the vectors. This

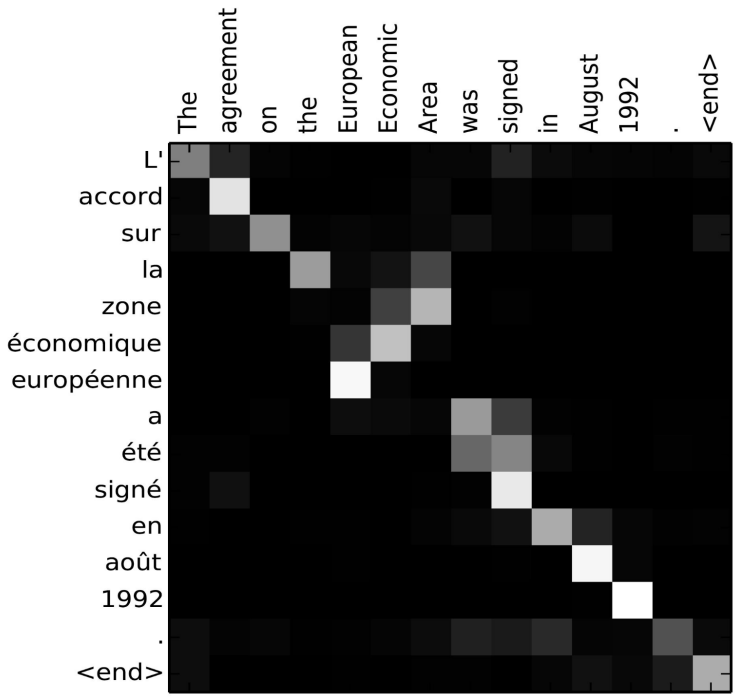


Figure 2.9: Example of the attention matrix for neural machine translation from English to France [6]. The increasing white saturation of the field means increased attention for given the word. It is apparent that the network performing the neural machine translation (NMT) task does not care about all the words in the input sentence but only a few relevant to the target word in the output sentence. This is the idea and the output of the Attention layers.

division serves as the countermeasure for more oversized dimensions, leading dot-products to larger magnitudes and softmax function to parts with slight gradients. As the last step, we perform the softmax function, which normalizes our values and gives the matrix of attention values for the pair of each two words. After this, one last step is to use this newly acquired matrix of normalized values and multiply it per row with the values. This gives us the weighted, attention-focused vectors for every token of each token in the Query.

There are different types of Attention. These types serve similar purposes but differ a bit based on their use case. We list a few of them below as an example for a better understanding of the idea behind the Attention.

- **Multiplicative attention**
  Calculates the attention weight by taking the dot product of the two vectors.
- **Additive attention**
  Very similar to Multiplicative attention, the difference is that additive attention applies a softmax function at the end.
- **Self-attention**
  Also known as transformer-based attention, it is a mechanism that operates on a sequence of inputs and assigns a weight to each input based on its similarity to other

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Q	Query,
V	Values,
K	Keys,
d_k	dimension of K and Q.

Figure 2.10: Mathematical demonstration of the attention functionality [41].

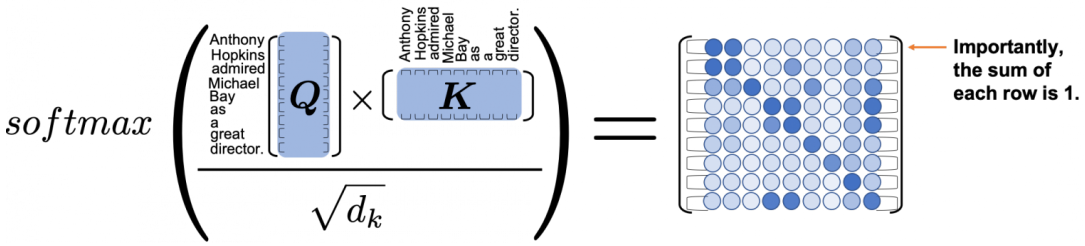


Figure 2.11: Example use of self-attention taken from the data-science blog [40]. The parameters Q, K, and V are all learned during the fitting process of the LM. Please note that in this figure is absent finale multiplication with the values of the tokens.

inputs in the sequence. It is handy for tasks that involve long sequences, such as machine translation and document summarization.

- **Multihead attention**
  This is a particular type of attention that splits the input query into multiple subspaces and applies different attention heads on each of them. The head's output is then concatenated and passed through the linear layer. We can interpret it as a kind of ensemble where we ask multiple smaller systems about the input query and get multiple opinions. We then aggregate these opinions with linear layers to get one attention value.

The main difference between these attention mechanisms lies in how they calculate the attention weights and the types of input sequences they are best suited for. Additive and multiplicative attention are often used for sequence-to-sequence models, while self-attention is commonly used in transformer-based models. Each attention mechanism has its strengths and weaknesses, and the choice of the mechanism depends on the specific task and the characteristics of the input data.

Chapter 3

Code Generating

In the previous chapter, we briefly discussed the NLP science behind Code generating and many other tasks that deal with human language. Now we will discuss this report's task; Code generating using textual description in human language.

We discuss some of the methods, phases, and optimization techniques the code-generating process uses. According to a paper by Yichen Xu and Yanqiao Zhu [42], in order to create a valuable system for code generating, we need to split this process into three sub-categories: Preprocessing, Sequential modeling, and Pretraining and finetuning. In this chapter, we go through each of them and describe the techniques used. We describe these steps from a more practical point of view, where we focus more on the code-generating task.

3.1 Preprocessing

The first step in creating our system is data collection and preprocessing. Data in this context mean the number of raw code files ready to be processed. The preprocessing usually takes these raw files and parses them into a more structural format, such as Abstract Syntax Tree (AST). Also, the preprocessing process can extract additional prior knowledge from programming languages.

One of the main tasks in this thesis is to create a usable corpus which means collecting and processing a big chunk of data and transforming it into a structural dataset. This process is more described in the practical section 5.1, but in theory, we can split this process into several steps:

1. Definition of the problem: This means to know what problem we are trying to solve and what data will help us achieve this solution. In our case, the problem is generating usable code in the CUDA language.

2. Identify relevant data: We need to acquire a source of data from which we filter and process the relevant data for our task.

3. Collect data: After identifying our data source, we need to collect the data. This usually means downloading public sources, web scraping, or perhaps collecting data from sensors.

4. The next step is to clean and preprocess the data to remove any errors or inconsistencies that could affect the quality of the training dataset. This may involve removing

duplicates, filling in missing values, or transforming the data into a standardized format.

5. Split the data into subsets (training, validation, and optionally test sets): After the data has been cleaned and preprocessed, it is typically split into three sets: a training set, a validation set, and a testing set. The training set is used to train the model, the validation set is used to tune the model's hyperparameters, and the testing set is used to evaluate the final performance of the model.

6. Normalize and standardize the data: Before training the model, it is often necessary to normalize or standardize the data to ensure that all features have a similar scale and distribution. In numerical datasets, this means, for example, normalizing the values to a specific scale. For text data, this means, for example, ensuring the same writing style or erasing typographical errors.

7. Data augmentation: This technique is not mandatory, but it helps with the training process. The data augmentation helps to achieve better generalization of the models' performance. One often used augmentation technique is putting random noise into the input data. For images, this can mean image distortion; for text, for example, shuffling or masking certain parts of the text.

Overall, data collection and preprocessing are essential steps in creating a high-quality training dataset for a neural network. These steps help to ensure that the data is clean, consistent, and well-structured, which can improve the accuracy and generalization of the trained model.

The techniques used in this thesis are described in the section about creating our corpus 5.1.

3.2 Sequential Modeling

We state ourselves at the point where we have our data. Now we need the model to train on our data. Sequential modeling refers to building machine learning models that are capable of processing sequential data and predicting what the next token in the sentence is. This means, for example, time series, music, video frame, or, in this case, code.

Even though we said that we are using transformer architecture in this report, we could also use, for example, Recurrent neural networks [29] with its variations LSTM [21] and GRU [11], or even Markov chains [13]. These models are older than the Transformer architecture published in 2017, and as with any other model, they have their limitations.

To further describe how models predict the next token in the sequence, we would be rewriting the mechanism behind the transformer, which is already described in its section 2.6.

3.3 Pretraining and Finetuning

Now, we have our data and choose the sequence modeling model we want to train. Next is the actual training process. Nowadays, we usually do not train our models from scratch, even though it is possible. Instead, the technique uses a pretrained model for the general purpose and finetunes it for our specific downstream task. This is because today's models are usually large (LLM) and require extensive long-term training on enormous amounts

of data. It is almost impossible to train a state-of-the-art model from scratch without significant computational resources and time. In this thesis, we also went down the road of finetuning existing models, which are further described in the practical section of this report.

Focusing on text generation, the training consists of inputting the input sequence into the model, generating a response, and comparing this response to the data samples' ground truth. Based on this comparison, calculate the loss value and perform back-propagation on the model.

This training process for text generation can differ a bit based on the performed task. Also, different kinds of training can help with generalizing. Some of the most used techniques for text generation are:

- **Masked Language Modeling (MLM)**
  Training process used for pretraining BERT. The idea of MLM is to mask some of the tokens in the input sentence and let the model guess the original tokens. It is similar to the 'Bag of words' process mentioned in section 2.5. According to a paper about AlphaCode [27], this process proved helpful for smaller datasets and the model understanding of the text.

- **Next Sentence Prediction (NSP)**
  Another method that originated from training the BERT model. The NSP method splits the input sentence into two parts, and after inputting the first part of the sentence into the model, we want our model to generate the second part.

- **Masked Span Prediction (MSP)**
  This method can be interpreted as combining the first two MLM and NSP. Instead of masking out single tokens in the input sentence, we mask whole spans, which the model is then asked to fill.

- **Unidirectional Language Modeling (LM)**
  Predicting the next token based on already seen/generated tokens. This approach is used for decoder-only architectures where we mask the tokens from the input sentence based on which token model is currently generating. We can further divide this process into the auto-regressive mode, where the model uses previously generated tokens to generate the next one, or the so-called Teacher-forcing mode, where the model does not consider the previously generated tokens but instead looks at the ground-truth tokens from the input sentence.

3.4 Sequence Decoding

The last thing we should mention before discussing the Code generation task is sequence decoding/searching. What it is, what types of searches there are, and its differences. This comparison and description are written accordingly to the study by Jin Yong Yoo and John X. Morris [43].

Code generation, like many other tasks in NLP, is a seq2seq task, which means that we feed our neural network with an input sequence (for example, tokenized sentence), and we want to generate another sequence accordingly to the input sequence and the performed task.

The model, however, generates probabilities for each token in its vocabulary, which can be tens of thousands of tokens for larger models. Another problem is that depending on the chosen token, we affect generating the rest of the sentence. Handling these probabilities and generating the best output response is called „sequence decoding," and there are several ways to handle it.

- **Beam Search**
  Very simple, although a very time and memory-complex algorithm. The beam search takes one parameter w, which we call beam-width. The beam search generates the w number of best follow-up tokens (tokens with the highest likelihood) for each next token in the output sentence. After generating ends, the beam search looks at the generated combinations, calculates the joint probability of the whole sentence, and picks the one with the highest value. This means that the likelihood value of the sentence is calculated as:

$$P(s) = \prod_{j=1}^{n_i} p(t_j)$$

  s Sentence.
  n_i Number of tokens in a given sentence.
  t_j Token of a given sentence.

  Figure 3.1: Likelihood calculation for beam search sentence.

  This, of course, gives us a significant number of combinations because the complexity is exponential $C = n^w$ where C is the number of combinations, n is the average length of a sentence, and w is the beam-width parameter.

- **Greedy search**
  This is a particular case of the Beam search where the beam width is equal to 1. This means that the greedy search takes only one most probable token for every newly generated likelihood value of the following output token. Therefore, the complexity is reduced to linear, respectful of the average sentence length.

- **Genetic algorithm**
  The first two mentioned methods for sequence decoding are analytical and deterministic, but the genetic algorithm is not. This method is not strictly defined and can have various behavior based on the definition and the input parameters. At the core, the genetic algorithm randomly generates a fixed number of solutions[1], takes the n of the best ones and combines them to achieve even better results.

- **Particle Swarm Optimization**
  Another stochastic algorithm for optimizing the space search is where a fixed number of members of the swarm search the distributed space between them in an effort to find the best global solution. The parameters for this method are the size of the swarm population and the number of iterations.

[1] Can computers generate truly random numbers?

3.5 Code Generating for Competitions

In this section, we would like to look at different points of view of coding assistants. We want to briefly mention competitive programming, where AI can challenge other computers and real programmers. These challenges are composed of complex tasks that the contestants must solve. The key parameters to grade the final solution are time $ memory complexity, code visual and structure, and more. This is an important part of this report, as it highlights differences between AI and human coding but also illustrates previously mentioned methods of training and evaluating. This section is mainly resourced by the paper from DeepMind [27] and study from Carnegie Mellon University [17].

We will highlight the AlhpaCode model as our representative of the code-specialized NLP models. The AlphaCode model is the last year's NLP code-specialized model trained specifically for competing with human programmers in the Competition-programming challenges. It is an Encoder-decoder transformer-based model that auto-regressively generates output.

Today's code-specialized models are not precisely working as human programmers. The central concept is similar as both the AI and the humans will look at the input of the problem, decide which parts are crucial for generating the correct solution, and generate the corresponding it; however, the AI has the unfair advantage, as we can see, for example, in the work of DeepMind laboratory and their model AlphaCode [27]. This model can generate hundreds to thousands of solutions and, based on the frequency of the solutions, pick the best one.

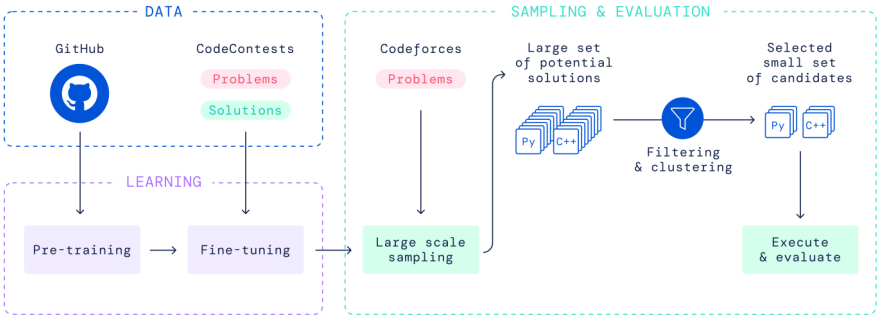


Figure 3.2: Diagram of AlphaCode training and inference [27]. The model has been pre-trained on the GitHub repository and finetunned on competitive tasks dataset from Code Contests.

Figure 3.2 shows the AlphaCodes training and evaluating process. The left part of the figure describes data and its usage. The right side describes how the model considers and samples the possible solution to a given problem.

Arguably, the most exciting part of the evaluation process is sampling and clustering, where the model has to cherry-pick only a few of the original solutions with the most considerable likelihood of being correct. This is where we can improvise and try different solutions. The AlphaCodes way of clustering fibs in executing the generated solutions and clustering based on the given answers by these solutions.

The interesting thing to mention is also the graph that the DeepMind published 3.3 that shows which techniques were used to optimize the training process, how beneficial they were, and what was the AlphaCode's final score in competitive programming challenges.

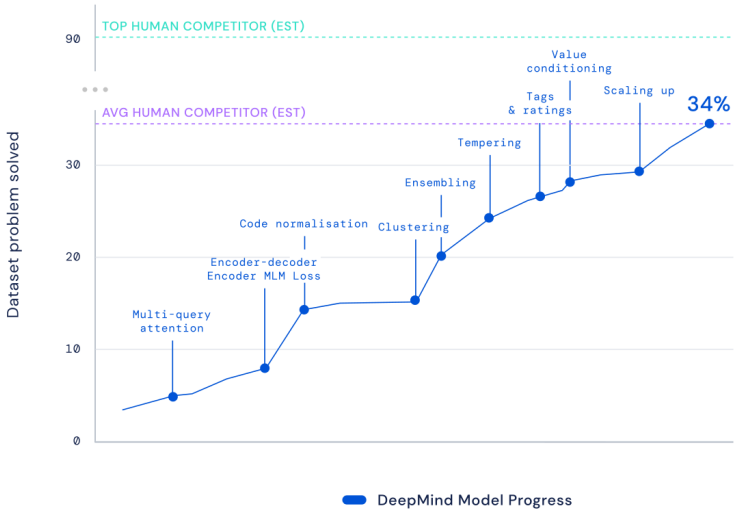


Figure 3.3: Graph of AlphaCodes performing development based on usage of different optimization techniques [27]. The graph score shows the percentile of human competitors that the network beat in programming challenges.

In this report, we want to highlight and discuss some of the techniques used as these are the techniques that can be used for the main task of this report which is generating CUDA code based on the text description.

- **Multi-query attention**
  This was already discussed in this report. It is the usage of attention heads during inference to highlight the most important input features. The *M*ulti-query means that the model uses multiple Attention layers.

- **Ensembling**
  This is equivalent to group programming in the AI world. Ensembling means we have multiple independently trained models competing or cooperating to find the best possible solution. In the case of AlphaCode, the Authors used several models of different sizes. We could argue that in the competition, several competitors were coding for one team.

- **Tempering**
  This regularization technique was used during the AlphaCodes training process and was first introduced by Dabre and Fujita [14]. It is a process of artificially modifying the probability distribution of tokens. This can smooth the distribution during training and lower the chance of over-fitting.

Chapter 4

CUDA Language

Before diving into the practical part of this report, it would be unwise not to talk about the CUDA language, which is an integral part of this thesis, as this defines the task we want to use in our model. We should mention the difference between classic mainstream programming languages like Python or Java and CUDA. To answer this question, we should first look at how GPUs work.

GPU or Graphical Processing Unit, according to academic slides from FIT BUT Faculty, which are resourced from NVIDIA web guides [5] and book by Kirk, David B, and Wen-Mei, W Hwu [23], is a massively parallel processor. The classic CPU comprises a few main parts: A control unit, Cache, RAM, and Computation Unit (ALU). The GPU has similar architecture but with far more computation units allowing the GPU greater parallelization. This difference is shown in figure 4.1.

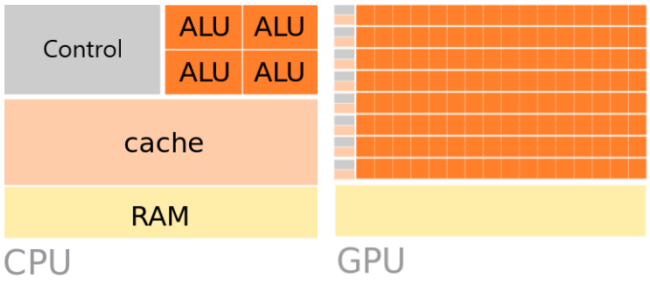


Figure 4.1: Comparision between CPU and GPU architecture, resourced from academic slides from FIT BUT Faculty. The GPU architecture has far more computation units than CPU and decentralized control units.

It is essential not to think about CUDA language as a stand-alone language but more as an extension of the C/C++ language. The programs that use functions implemented in CUDA are usually closely connected to the central control on the CPU side written in C/C++.

The first difference we want to address between CUDA and other languages is that the programmer must always bare in mind that the code he is writing is in parallel. That means that several threads execute the instructions at once. This means that the GPU code is being executed at massive parallel and, therefore, can process more data in the same unit of time as the CPU.

However, this computation advantage does not come for free. The problem with greater parallelization is lower control over data coherence in RAM. Another problem with GPU is that the code has to have the least amount of branching possible. That is because the GPU's instructions are called SIMT or (Single Instruction Multiple Threads). This means that every instruction from the code executes multiple threads simultaneously. The problem happens when the code and, therefore, the threads get branched. If that happens, the instructions in different branches must be executed sequentially, and consequently, the execution loses valuable parallel power. This phenomenon is illustrated in figure 4.2.

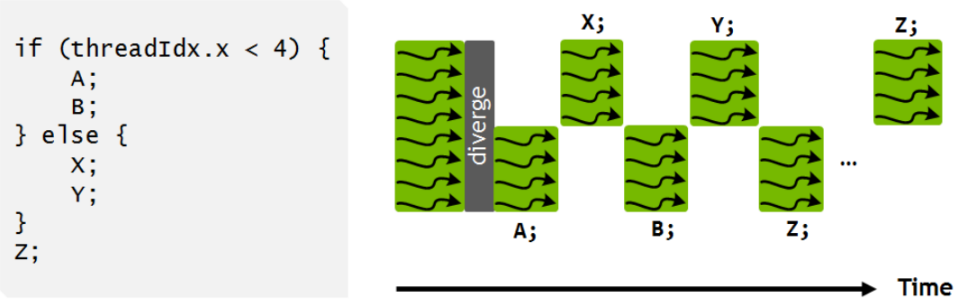


Figure 4.2: example of how the execution time is worsened when using code branching [3]. After branching on the 'if' statement, the execution is divided into two halves (2 branches), and both branches are executed sequentially.

Another difference with GPU code is explicitly choosing the type of memory in which the data should be stored. In contrast to the CPU, the GPU has several types of memory, which are on the GPU chip:

- **Global memory**:
  This memory is shared across all threads and is read/write. Access to it is, however, very slow.
- **Constant memory**:
  This memory is global for all threads across all Streaming Multiprocessor (SM) blocks. GPU threads cannot modify the data. Access to this memory is high-speed.
- **Texture memory**:
  Very similar to Constant memory, but the data here can be arranged into dimensional blocks for easier access.
- **Shared memory**:
  This memory is shared exclusively with threads from the same SM blocks. GPU is usually a compound of SM blocks that contain threads. Every SM block has its shared memory. This memory is read/write and is very fast. The usual use of this memory is by storing often-needed values in it at the beginning of the executed kernel and using it there.
- **Local memory**:
  Local memory is global memory with interleaved addressing, which makes iterating over an array in parallel faster than having each thread's data blocked together. This memory serves as a last resort for threads when running out of registers and shared memory.

- **Registers**:
  The first and only memory that is exclusive to every single thread. It is read/write and very fast. In the code perspective, every variable/constant that is defined inside the kernel and is not shared memory is stored in registers or local memory.

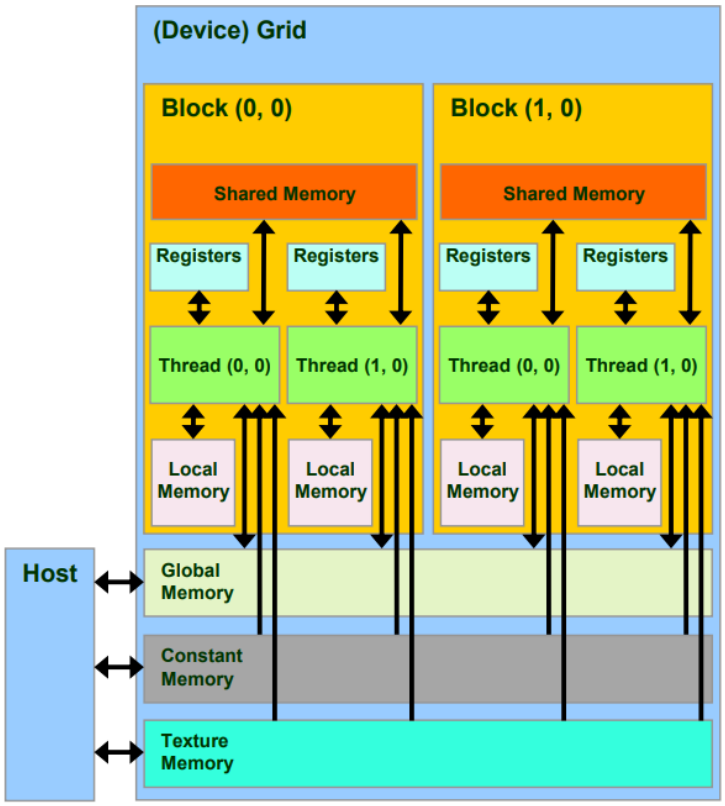


Figure 4.3: Diagram of GPU architecture. The source of this figure is from official academic slides for FIT BUT faculty. The diagram shows the arrangement of threads into blocks and the data flow of every type of memory present in the GPU. The host is CPU and RAM.

Another thing the programmer must be aware of is how to define the CUDA kernel. We already said that the CUDA code is syntactically the same as the C/C++ code. The CUDA, however, uses unique keywords/prefixes to identify its kernels.

- **___global___**
  This kernel prefix tells the compiler that this kernel is defined on the device (GPU) and can be called from the host (CPU) side.

- **___device___**
  The device prefix indicates that this kernel will be defined on the device side and cannot be accessed by the host.

- **___host___**
  The host prefix is the opposite of the device prefix. It hints to the compiler that the code should be defined on the hosts' side and cannot be accessed from anywhere else. Note that if the kernel has only the host prefix, it is functionally the same as if it had no prefix at all.

The CUDA kernel prefixes can be combined, and their functionality remains the same. This means that, for example, the kernel that contains ___*device*_____*host*___ prefix is defined on both device and host side but cannot be called cross-platform.

One thing that the ___*global*___ kernels differ from the other types is that we can see these kernels as a starting point where the GPU executes defined logic. This means that from this point starts hundreds to thousands of threads. Therefore, it is crucial to identify these threads to keep better track of what each thread is supposed to do. We do this by calculating its ID within the GPU grid context. From the GPU architecture represented in figure 4.3, we can see that the GPU is divided into blocks, and within these blocks are threads. These blocks and threads can be further composite into the 3D structure. To calculate the global ID within the grid (GPU), we need to know the dimensions of these 3D structures. Fortunately, CUDA provides us with the global variables **threadIdx**, **blockIdx**, and **blockDim**, which we can use for this purpose.

```
int idx = threadIdx.x + blockIdx.x * blockDim.x;
```

Figure 4.4: Calculating grid idx for each thread in the kernel. Note that this code assumes a one-dimensional grid and block configuration. If you are working with two or three-dimensional grids, you will need to use **threadIdx.y** and **threadIdx.z** for the thread ID and **blockIdx.y**, **blockIdx.z**, **blockDim.y**, and **blockDim.z** for the block ID and block dimensions. The global ID calculation will also need to be adjusted accordingly.

Another important aspect of the CUDA language is CUDA primitives, which are simple functions defined in the CUDA core that are used for simple tasks. We can list ___*activemask*, ___*shfl_down_sync*, or ___*ballot_sync*, but probably the most important ones are ___*syncwarp* and ___*syncthreads* primitives that serve to synchronize threads within the SM machine and within the running warp of threads. The behavior of these primitives is similar to barrier pragma in OpenMP, where all the running threads stop on this barrier and wait until all of the threads reach the stop. The ___*syncwarp* primitive mimics this behavior and applies it to the currently running warp in the SM machine (usually 32 threads). It should be noted here that ___*syncwarp* is not necessary for every GPU architecture as some of the architectures do not support thread divergence within the warp.

If ___*syncwarp* works on warp level, then ___*syncthreads* work on the block (SM) level. This means that this primitive synchronizes all the threads within the current block. The number of threads can differ based on the programmer and the parameters the ___*global*___ kernel has been called with.

To summarize this section, how programmers think when implementing programs in the CUDA language has to be different from other languages. Commonly the implemented functions have to identify each thread by calculating its global ID based on the thread ID in the block, block ID, and block size in the grid. The programmer has to know which memory to use and how to use it because the access patterns to different memory types can be resolved in different performances and even errors. The topic of coding in the CUDA language is quite comprehensive, and there is, of course, more to it, such as graphs and other intermediate techniques. We also completely skipped the data transfer between CPU and GPU, which would be enough for another section. However, this brief introduction to the practical coding guide in CUDA language is sufficient for this thesis.

Chapter 5

Design and Implmentation

Up until now, we stayed on the theoretical level and introduced the reader to the field of NLP, machine learning, and CUDA. We covered most of what the reader needs to understand in this next practical part, where we talk more about a practical approach to this problem. We propose several solutions, analyze them and discuss their results. In this chapter, we discuss the process and pipeline of creating and analyzing our own dataset for language model finetuning, we introduce several transformers we used for finetuning, and we analyze the models' results and discuss sources of errors and possible improvements.

5.1 Data

Probably the most significant and important part of this thesis is the process of creating and validating our dataset. As mentioned previously, current language models do not specialize in coding in CUDA, not even those that are designed for coding, and as a consequence of this, there is an absence of valuable and relevant corpora to be used. Therefore, this section describes our process of creating a new dataset. We introduce our pipeline and explain how this pipeline can be used for creating almost any code-generation corpus. We also discuss the methods we used to clean up the corpus and the performed analyses.

This section is split into multiple subsections. The Data gGathering 5.1.1, where we discuss how we collect the data and the size of the corpus, and the Data Processing 5.1.2, where we describe the cleaning and validation process of the data. The next section is then Corpus analyses 5.1.3. This section is devoted to performing analytical methods onto the created corpus in order to find out important metadata that helps us to find out the dataset quality. The last section is then Repository Analyses 5.1.4. As mentioned in the Data Gathering section, we used GitHub as the primary source of the data, and analyzing the repositories we used can be a good indicator of code quality.

5.1.1 Data Gathering

This section is about the data gathering, e.g., how we collected the crafted corpus, how much data we collected, and how this system can be used for further use. We used GitHub as the primary source of our data and we mimic the approach used in the paper from Lars Bjertnes and Jacob O. Tørring and Anne C. Elster [8] about creating Autotunning dataset for CUDA. The authors used googles Big Query[1] to access the GitHub log archives and

[1] BigQuery Website

find CUDA repositories to download and process. We analyzed the GitHub archives from the years 2019-2022. Note that Google GitHub log archives are accessible all the way up to 2015, but we decided to exclude the rest of the year, because of the software lifetime and possible use of deprecated functions and libraries.

We used Big Query's built-in SQL query API to obtain all the logs where the target repository included "cuda" (case-insensitive) in the name. After dropping the duplicities, we obtained $\sim 26,000$ repositories. After we got our list of repositories, we implemented a script that tried to download each repository locally. The result was approximately $\sim 13,000$ repositories stored on local machine. Besides that, we also download metadata through the GitHub API using links queried from BigQuery. The collected metadata about the repositories is used later in the experiments.

After we collected our repositories, we implemented a parser that recursively searched through the repositories and found the relevant files and saved the found CUDA kernels and corresponding metadata to the database. Note that we restricted the parser to search only for CUDA kernels, as this is the primal focus of this thesis. We included kernels with at least one of the CUDA prefixes mentioned in the section about the CUDA language 4.

Note also that we cannot include names of all of the repositories in this report as there are too many and so we include the list in the publicly available GitHub repository[2]

As our storage system, we decided to use MongoDB[3], which is a NoSQL document database. The created dataset consists of 5 collections and can be represented, as the simplified version, by a UML diagram:

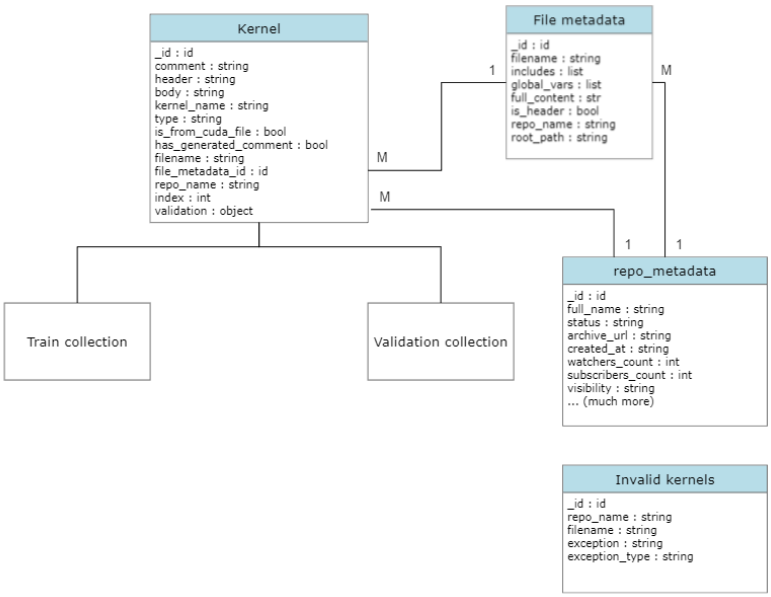


Figure 5.1: Simplified representation of the database structure in MongoDB. The *repo_name* collection consists of all metadata we acquired from the GitHub API. The CUDA kernels are split into training and validation parts based on randomness and certain parameters described later.

[2]Used repositories.
[3]MongoDB Website

5.1.2 Data Processing

The previous section about data collection described how we collected our raw data 5.1.1. We mentioned that we restricted our data collection exclusively to CUDA kernels/functions. The data processing section elaborates more on how the CUDA kernels were cherry-picked from downloaded repositories and how they were evaluated in terms of static analysis.

We proposed a parser script that recursively iterates through the repositories, reads files with defined suffixes, and detects and collects individual kernels. Each of the kernels then performs simple cleaning such as adjusting indent, transforming comments to use the `//` prefix instead of `/**/` or stochastically generating a comment of the kernel based on the defined header. Other methods used for cleaning the kernels were, for example, removing the namespaces and html/markdown/xml tags in comments or clearing indentation.

Parser then inserts these kernels into the training or validation part based on the defined split ratio and other parameters, such as machine-generated comments where these kernels were automatically put into the train section.

Besides the individual kernels, we also included several other collections represented on the UML graph 5.1. We included file metadata for each file that included at least one CUDA kernel or was considered a header file based on the suffix.

The file suffixes for the parser to search were defined as *c, cc, cpp, cu, h, hc, hcc, hu*:

The algorithm to detect the CUDA kernels in the file was built upon using regexes and worked in two iterations mode.

In the first iteration, the algorithm detected the CUDA function headers based on prefixes *___device___*, *___host___*, *___global___*. Note that in some repositories, they used macros for defining *host, device* prefixes together; this issue had to be addressed and detected as well. These headers were parsed and saved with their start and end file indices.

The second iteration went through all the detected headers, and each tried to parse the body and possible comments. The function body is a mandatory function, as the bare declaration of the function is unsatisfactory. The comment, on the other hand, was, when absent, generated based on the kernel header.

After the second iteration and collecting the CUDA kernels, the parser iterates through the file one more time. It collects additional data, such as included libraries in the file, global variables, macros, and others. All of this data are then stored in the database.

We mentioned that we collected roughly 13,000 repositories based on their name. Unfortunately, not all of them had to do something with CUDA language. For example, repositories from the user named "*baracuda*" had been included in the pack, even though the repository had nothing in common with CUDA language. After analyzing the extracted kernels, we found out that from the original $\sim 13,000$ repositories, only $\sim 7,000$ were used.

We managed to extract approximately $\sim 490,000$ kernels from the downloaded repositories. In spite of the fact that this is a fairly large number, a lot of this data comes from test repositories of various users with no guarantee of code quality or functionality. Another step is to analyze and evaluate the quality of the processed code and repositories themselves. This is a part where the **file_metadata** and **repo_metadata** collections in our database came to play. We used the collected file_metadata to evaluate kernels in terms of syntax correctness, and we used repo_metadata collection to sort and prioritize repositories based on certain measures that are further described in the Repository Analyses section 5.1.4

5.1.3 Data Analyses

As mentioned, our goal is to make a helpful corpus that can be used for finetuning large language models for coding in CUDA language. We already described how we collected and processed the data to make our corpus. This process allows its user to download data from GitHub and transform it into a usable dataset. However, this data could be noisy and does not necessarily produce the best results. We need to filter and prioritize more quality kernels than others. This brings out a few questions. How do we find if a kernel is well written from the syntactic and semantic points of view? Furthermore, by what parameters do we prioritize and sort kernels?

We propose a system where we rank our kernels based on the syntactic analyses and kernels repository metadata. The syntactic analysis is based on if the kernel can compile into the machine code, what CUDA prefixes the kernel use, and if the code uses *___shared___* memory or not. We described the compilation process with diagram 5.2. This process tells us if the kernel is syntactically correct or, if not, then how faulty it is. The use of shared memory gives us a rough heuristic about the code quality in terms of performance. Generally speaking, we can say that CUDA kernels that use local and shared memory are more optimized than those that don't.

First, we describe the process of kernel validation. We should note first that for every repository, we used the master branch as it has the highest chance of consisting of functional and commented code. We could have missed some features and new functions present in other branches, but this additional content comes with a risk of corrupted code. Another thing to note is that there certainly is a possibility of trying to compile the whole repository to see if present kernels are valid; however, this approach was made by the authors of a paper about creating an Autotunning dataset for CUDA [8]. In their paper, they described that this approach comes with complications of complex make-files, and after their experiment concluded that the average rate of successfully compiled repositories out of the box is around 3%, which does not give us any useful information about the code quality.

As the alternative, we implemented a *cuda_kernel_validator* script that iterates through the kernels in the database and tries to compile them individually with additional content based on the given error from the **nvcc** compiler. The validation process for each kernel was divided into iterations, and each iteration was saved to the database together with the corresponding kernel. The evaluation process is represented by the BPMN diagram 5.2.

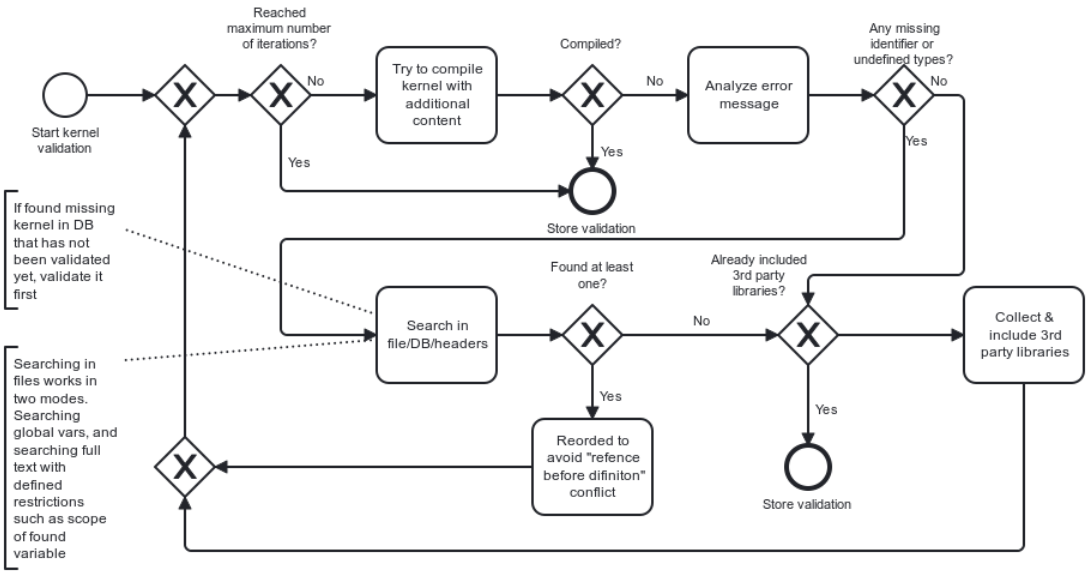


Figure 5.2: Representation of validation process for CUDA kernels.

Even though the validation process was built with the idea of searching thoroughly and finding most of the missing tokens, please note that there was still room for error. Further analyses of this phenomenon are described in section 5.1.3. Another metric for our data quality was based on the repository metadata. We collected through the GitHub API metadata for almost all of the downloaded repositories and sorted them. This is further described in section 5.1.4

On a full scale, the dataset consists of almost 500k kernels split into train & validation parts in a ratio of 92/8. We decided to use a smaller validation part due to the time concerns of the model evaluation, which is performed in an auto-regressive mode, unlike the training where we use teacher-forcing. This difference can significantly increase the evaluation time as the auto-regressive inference cannot run in parallel.

We ran our *cuda_kernel_validation* script for the whole dataset, and the compiled ratio came out to be roughly 40/60. This means we could not compile more than half of the kernels. We recognized several errors based on the prompt that the compiler returned.

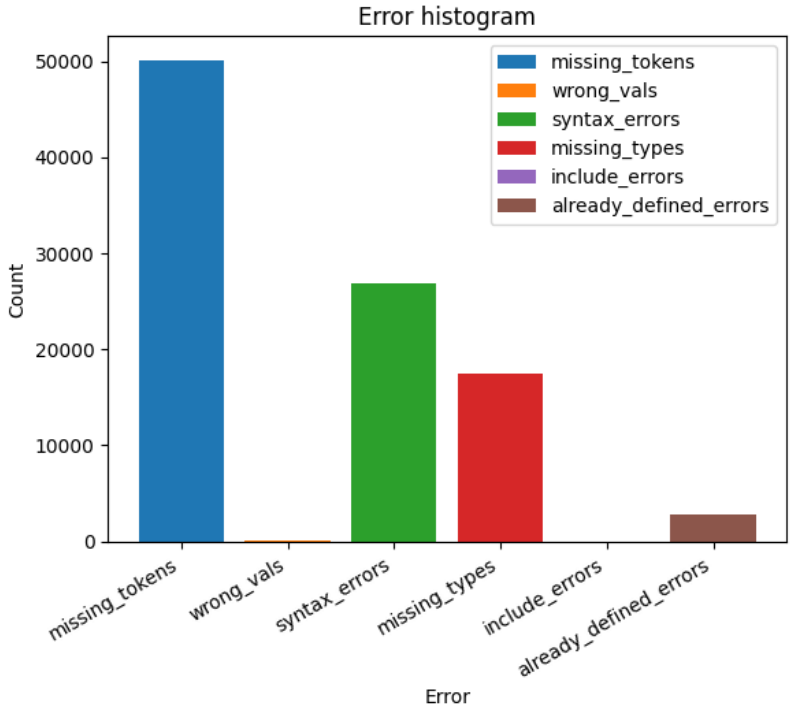


Figure 5.3: Histogram of classified errors in the dataset from a total of 486,414 CUDA kernels. We counted only one error per type per kernel. This means that if the kernel had more than one error of the same type, we ignore the rest of the same-type errors in the histogram.The legend of errors is defined here 5.3

- **missing_token**
  This error represents missing variables, constants, functions, or some interface the kernel is working on.

- **wrong_val**
  Short for wrong value. This error type represents that the wrong value has been initialized or set to some variable.

- **syntax_error**
  Syntax errors typically mean errors on the syntax analysis level when the syntax tree cannot be put together due to an invalid sequence of tokens. This means, for example, missing semicolon, unclosed parathesis, and other...

- **missing_type**
  This error is specific for cases where the compiler misses the declaration of the variable. This means that in the code, there is a variable that has not been initialized.

- **include_error**
  Error typical for including a library that does not exist.

- **already_defined_error**
  As the name suggests, this error type represents cases where one variable is initiated more than once,

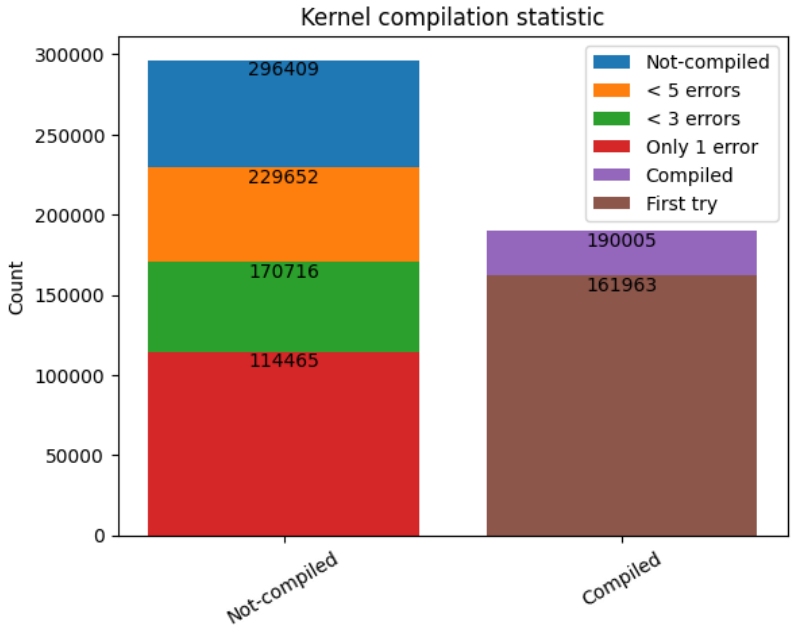


Figure 5.4: Compiled ratio statistics based on final compilation state and the error analyses from the last iteration of CUDA kernel validation. The total number of 486,414 CUDA kernels. In the **Not-compiled** bar, we divided the bar into several categories based on a number of errors from the last compilation of the kernel. The **First try** section in the **Compiled** bar represents kernels that were compiled without any other dependencies to be found.

The 40/60 split is not ideal, and from the figure 5.3, we can see that regardless of the implemented script designed to search for the dependencies, the most frequent error in the corpus was missing identifier or token. We sorted the repositories based on the number of errors and looked at the most faulty ones. Then we grouped *missing_token* errors by the token name and looked at the most occurring identifiers. From the analyses of kernels where these tokens were missing, we found that most of these tokens did not come from the kernel file but included libraries that did not match the same version the authors of the repositories used and therefore did not contain the same name functions and wanted properties. Due to this error, the majority of the recorded errors were outside of the test CUDA file. After this discovery, we decided to recount the errors the same way as shown on graph 5.3 but this time filter errors outside the CUDA kernel file.

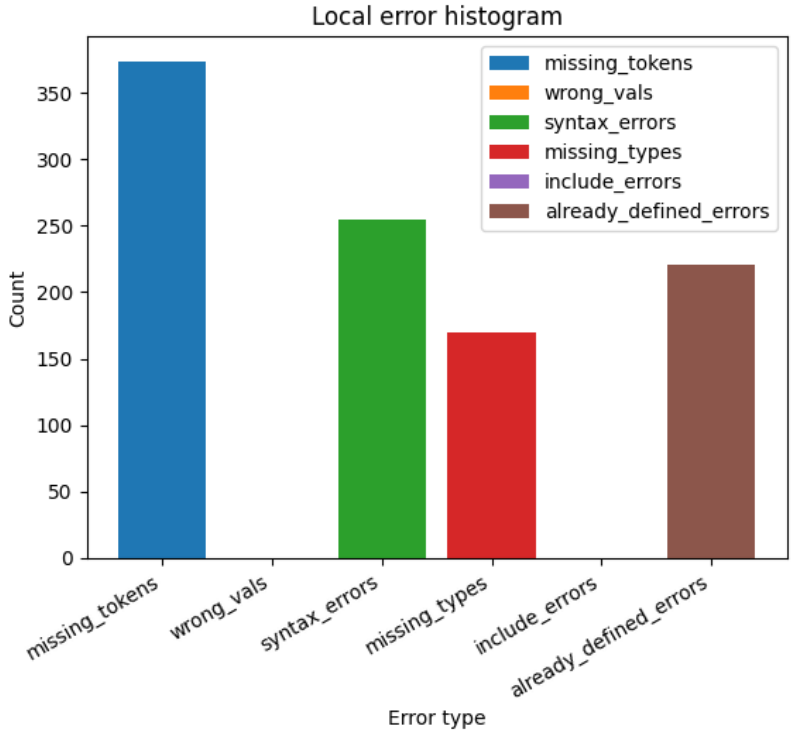


Figure 5.5: Histogram of classified errors in the dataset from a total of 486,414 CUDA kernels. Excluding errors that happened outside the were found outside the CUDA kernel file. Compared to the graph 5.3, the error count is several orders of magnitude lower.

The discovery that the library files are causing most of the analyzed errors brings problems to the corpus analyses. It would take significant work to automate the process of finding a fitting library set for each repository and load them dynamically. From the figure 5.5, we can see that errors that happened in the created test file with the tested kernel are far less compared to the total number of errors shown in the figure 5.3. We will assume that most of the kernels in the dataset are syntactically correct, and we filter only those we found as faulty from the performed analyses.

We also mentioned in the Data Gathering section 5.1.1 that we were generating comments for the CUDA kernel without one. This can significantly skew the results both ways and we should mention that out of the total 486,400 kernels, only 95,116 (19.56%) had an original comment. The structure of the generated comment was designed to mimic doxygen comments and therefore copy the most usable comment structure nowadays.

Another thing we did was group the kernels based on their CUDA prefixes ___*device*___, ___*host*___, ___*global*___. We found 17,680 kernels that were defined only for the ___host___ side (CPU). These kernels do not have functionally almost anything in common with CUDA language besides the used prefix. However, we decided to keep them in the dataset as they still represent valid C/C++ code. The total number of each variation of these CUDA prefixes is shown in figure 5.1.

We also analyzed the CUDA kernels in the corpus for using the optimized ___*shared*___, ___*constant*___ memory. We know that the use of these memories depends on the implemented algorithm, and it is not always beneficial to use them; however, we still feel like their use is recommended and indicates a higher quality of code. From all the kernels captured, only $\sim 38,000$ uses optimized memory. Most of which are ___global___ kernels

which are to be expected as the ___global___ kernels usually deal with the allocation of the ___shared___ memory at the beginning of the kernel's functionality 5.1.

Kernel type	Count	Optimized memory count
___device___	156,027	5,569
___device_____global___	180	4
___device_____host___	182,943	22
___device_____host_____global___	70	2
___global___	128,918	32,375
___host___	17,618	3
___host_____global___	72	0

Table 5.1: Grouped CUDA kernels based on their prefixes and use of optimized memory *___shared___*, *___constant___*.

We analyzed the lengths of the kernels we obtained and assessed whether our models, which have limitations due to time and memory constraints, can handle up to 600–1000 tokens. Our goal was to determine if these restrictions were adequate for our dataset. To better understand the data, we utilized a GPT2 tokenizer to convert the inputs into tokens and determine the actual input sizes for the model. We also calculated statistics on the collected data. We then entered this data into the table 5.2. We also created a histogram of the individual categories (comment, header, body, total). These histograms are available in the appendix A.1.

		Comment	Header	Body	Total
Mean	Raw	118.12	120,05	619.48	859.65
	Token	42.12	48.97	301.85	394.94
Median	Raw	75	71	94	284
	Token	27	24	50	119
Modus	Raw	58 (10,288)	59 (16,971)	2 (20,930)	172 (3,143)
	Token	27 (18,325)	19 (40,747)	2 (21,785)	73 (5,596)
Min	Raw	3	15	2	37
	Token	2	5	2	13
Max	Raw	13,754	11,514	2,538,338	2,538,563
	Token	5,690	6,032	1,735,140	1,735,245

Table 5.2: Length statistics over the obtained data. The "Raw" column represents untokenized data. The "Total" row represents the whole kernel with comment, header, and body. The bracketed numbers in the "Modus" rows represent the number of occurrences.

5.1.4 Repository Analyses

As we mentioned, we decided to analyze and sort our dataset using information about the downloaded repositories from GitHub to determine their quality. This metric can serve as an indicator of the quality of code from that repository. We decided to use three counts from the downloaded data:

- **Watcher count**
  This count indicates the number of users who have chosen to receive notifications when changes are made to a repository. Because of this, it can serve as a metric to understand the level of activity and involvement around their project. It can also be an indicator of the quality and popularity of the code.
- **Subscriber count**
  The subscriber count is similar to the Watcher count, with the difference that subscribers can choose what kind of news they want to be notified of on the repository. It serves an almost identical purpose for our use case, even though some may argue that the watcher count represents people with higher passion and activity for the project.
- **Stargazer count**
  The stargazer count is more of a passive metric and works similarly to the like button on social media. People who are stargazers might get news about the project. Still, mainly the star count works as a bookmark for the users and a rough indicator of the project's popularity in the GitHub community.

We decided to use a weighted sum for these metrics as we feel like the stargazer count is a bit less important than the other parameters. This means we weighed 1 for the watcher and subscriber count and 0.8 for the stargazer count. Unfortunately, we could not download over 2,000 repository metadata because GitHub API could not find these. We cannot say why we could download the repository archives but could not download metadata to them. Due to this error, we got a total of 32,889 kernels in training & validation sets that have no repository metadata. We have to exclude these kernels from this analysis. As the representation, we show the first five repositories with the highest score in this report.

Idx	Name	Watcher count (weight 0.8)	Subscriber count (weight 1)	Stargazer count (weight 1)	Total score
1.	kaldi-asr/kaldi	12,604	702	12,604	23,389.20
2.	NVlabs/instant-ngp	11,971	181	11,971	21,728.80
3.	isl-org/Open3D	8,440	172	8,440	15,364.00
4.	catboost/catboost	7,068	193	7,068	12,915.40
5.	cupy/cupy	6,820	129	6,820	12,405.00

Table 5.3: Five most valuable repositories by the metric of a weighted sum of given parameters.

We wanted to see if there is a correlation between the repository popularity and the number of CUDA kernels. This would be a good sign for us as we would be almost guaranteed to have high-quality data. We ran a statistic and discovered that the ten highest-scored

repositories contain only 2% of the total data, the top 20 repositories contain almost 3%, and the top 100 repositories, where the 100th repository has scored only a total of 44.2, contain 5.43%. We run a Pearson correlation test, and with a score of 0.08, we found out that, unfortunately, the hypothesis of correlation between repository popularity and its size is false.

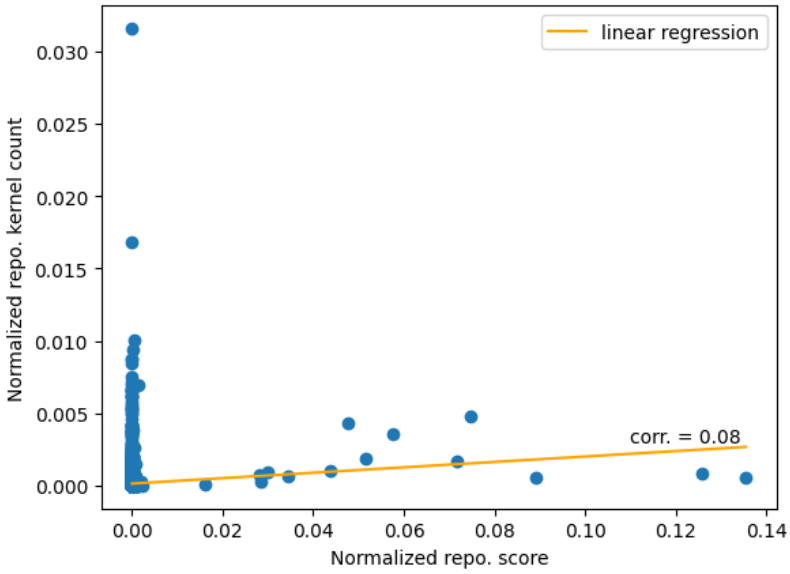


Figure 5.6: Scatter graph of the correlation between repository popularity, according to the GitHub metrics, and its size.

5.2 Design

We have already mentioned several times the use of the transformer models as the backbone of our research. In this section, we want to describe the design and structure of our implementation. In the previous sections, we discussed the process of gathering and processing data to create our corpus. This corpus is used for training/finetuning selected models, which are then evaluated on multiple metrics. This pipeline can be represented as the BPMN diagram shown below 5.7.

We can divide this diagram into two halves. The first, the upper half, represents the described process of gathering and processing data. This process is independent of the rest of the diagram but is an essential part of the design. After the first half is finished, the lower part of the diagram represents the training & evaluation of the models. Note that each model has a slightly altered process, for example, in creating mini-batches, but the diagram covers all cases from the abstract view. After each epoch, the process saves model weights together with the optimizer, average loss, and scheduler stats. This gives the user possibility to interrupt and resume the process as needed. After the last epoch, the evaluation process starts. Again, the evaluation process differs based on the trained model. For most cases, we used **pipeline** class in the **transformers** Python3 library with seed set to 1; however, for example, for the baseline model, we wrote our own sequence Greedy and Beam decoders. The pipeline class did not support all of the models, and we had to use the built-in “generate” function.

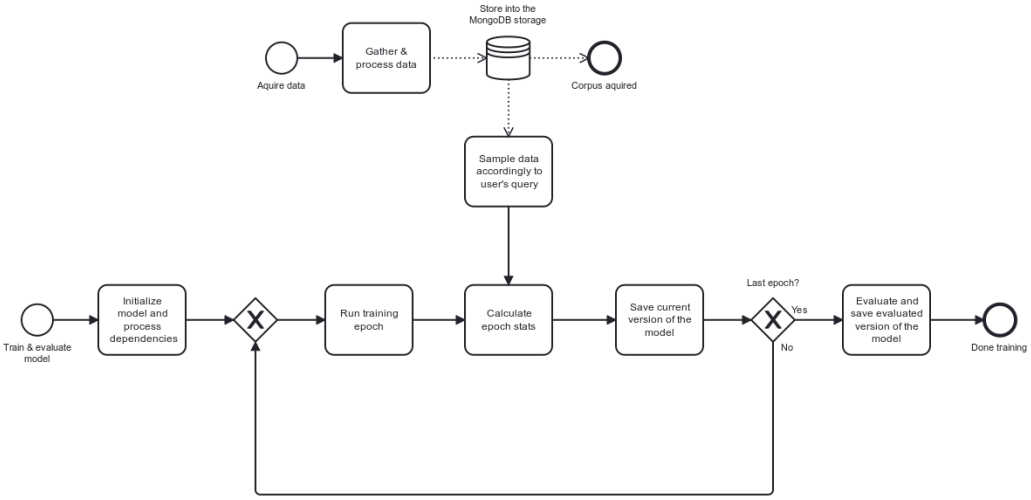


Figure 5.7: BPMN diagram representing the whole pipeline of our research.

The evaluation process evaluated the validation part of the dataset, calculated the BLEU score as the indicator of the model's performance, and saved the input prompts, target outputs, and models' predictions. We did not calculate the Rouge score and other metrics in the evaluation process because these metrics took significant time to compute. Computation of these metrics was executed separately after the training & evaluation process. For the same reason why we did not calculate all the metrics, we did not evaluate our models on regular bases during training. The evaluation and decoding of output sequences is very time demanding, and the process would prolong itself significantly.

5.3 Training & Evaluation

In this section, we want to talk about the training & evaluation process. Of course, this process differs a bit on an experimental basis, but we can still describe the common parts. We used different models to cross-validate the training results on our hand-crafted corpus and to determine if the corpus is usable for training/finetuning language models.

Models we used were taken from the Hugging-face community[4], and the training and validation script was written in Python 3.8 using the Pytorch framework[5]. We used a standard AdamW optimizer and schedule with warm-up duration to optimize the training process and evade early overfitting and getting stuck at the local minimum based on foremost training pairs. For every model, we used the cross-entropy loss function to compute the criterion. The list of all the used libraries for training, evaluation, and analysis processes is listed in the code repository.

This report focuses on the code generation from the **textual** description of functionality. However, because a lot of the acquired data had machine-generated comments, we decided to compose our samples into input prompt and output target, where the input prompt consists of the kernel's comment and header, and thy target consists of the kernel's body. This approach differs slightly from the Decoder-only architectures where the input prompt

[4] https://huggingface.co
[5] https://pytorch.org

is the whole kernel, and the output target is the same sequence shifted one token to the right.

We used teacher-forcing for our training process to speed up the duration, but we used auto-regressive mode for evaluating the model. Due to the time concerns, the evaluation was performed with greedy search (beam search with wide = 1) as it is the fastest search available. Please note that we are aware of the risk of using strict teacher-forcing for text generation as it is known from several papers [18], [25] and the 10th chapter of the deep-learning book by Ian Goodfellow and Yoshua Bengio and Aaron Courville [15]. We used teacher forcing mainly for the availability of parallelization and, therefore shorter training cycle.

Another thing we also tried, besides training the models from scratch, is to use pre-trained models on various tasks, following the idea that ̈language models are few-shot learners ̈from paper [9]. These experiments are, of course, further described in their sections.

5.3.1 Used Metrics

As the evaluation metrics, we used BLEU, Rouge, and Bert metrics together with hand-crafted metrics described below:

- **Bilingual Evaluation Understudy score (BLEU).** [34]
  BLEU score is a precision-focused metric that calculates the n-gram overlap of the reference and candidate texts. This n-gram overlap means the evaluation scheme is word-position independent apart from n-grams' term associations. The BLEU score is arguably one of the most common evaluation metrics for text-generating models. The BLEU score is in the original paper defined as:

$$p_n = \frac{\sum_{n-\text{gram} \in \text{candidate}} \text{Count}_{\text{match}}(\text{n-gram})}{\sum_{\text{n-gram} \in \text{candidate}} \text{Count}(\text{n-gram})}$$

Figure 5.8: Example of calculating the BLEU score [34]. An illustration of this equation can be found on figure 5.10. The fraction represents the total number of matching n-grams occurring in reference and candidate text over the total number of n-grams in reference.

- **Recall Oriented Understudy for Gisting Evaluation (Rouge).** [28]
  Rouge is a very similar metric to the BLEU score; however, the Rouge metric is recall-focused, as the BLEU score is precision-focused. The original paper has multiple versions of the Rouge score: Rouge-N, Rouge-L, Rouge-W, and Rouge-S. In our evaluations we use Rouge1 and Rouge2 precisions.

- **Bert score.** [44]
  The basic idea behind BertScore is to compare the contextual embeddings of the generated or translated text and the reference text at the word and sentence levels, using cosine similarity. The metric considers both the content overlap and the linguistic quality of the generated text and is particularly effective at capturing subtle differences in meaning and fluency.

  At the word level, BertScore computes the cosine similarity between the embeddings of each word in the generated text and the closest matching word in the reference text

$$P_{Rouge-1} = \frac{\sum_{(a,b)\in A} \text{match}(a,b)}{\sum_{(a,b)\in A} |a|}$$

$$P_{Rouge-2} = \frac{\sum_{(a,b)\in A} \text{match}(a,b)}{\sum_{(a,b)\in A} |a| - 1}$$

Figure 5.9: Equation for calculating the Rouge score between reference and candidate text [28]. In both equations, A is the set of all pairs of reference and candidate sentences, $\text{match}(a, b)$ returns 1 if the a and b match, and 0 otherwise. $|a|$ is the length of sentence a. An illustration of this equation can be found in figure 5.11.

based on the contextual information provided by the BERT model. The overall word-level score is the weighted average of the individual cosine similarities, with higher weights assigned to words that are more important for the text's overall meaning.

At the sentence level, BertScore computes the F1 score between the embeddings of the generated and reference sentences based on their common n-grams. The F1 score measures the harmonic mean of precision and recall and provides a balanced assessment of the generated text's overlap and uniqueness. This process is represented by the figure from the original paper 5.12

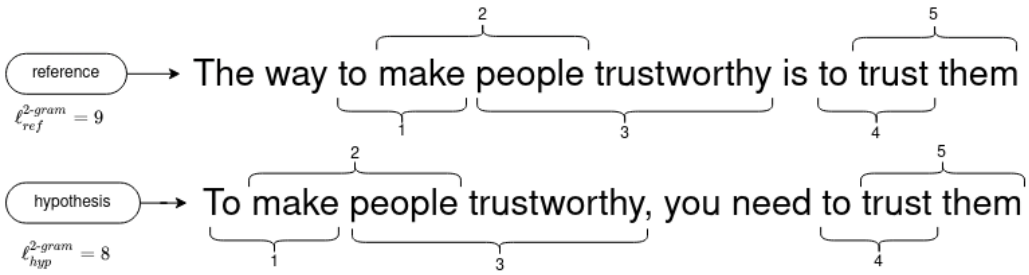


Figure 5.10: Example of calculating the BLEU score for given reference and candidate text [10]. The BLEU score can calculate its value based on the size of searched n-grams. For 1-gram the total value is 7/9, for 2-gram it is 5/8, and for example for 4-gram the BLEU score is 1/6.

We know these metrics will not give us an accurate benchmark of model performance. The code can be written in various ways and still execute the same task; however, these metrics were chosen as one of the best language metrics to benchmark the seq2seq task we have today.

Apart from the mentioned metrics, we also created a script that will attempt to compile and evaluate the model-generated code based on the error message.

The script will take generated code of the model together with the kernel header and tries to compile it with additional content generated when validating the original kernel. The BPMN diagram below represents this process:

The idea behind this metric is to analyze the syntax correctness of the generated code. We are aware of the lack of semantic validation and the total absence of validating if the

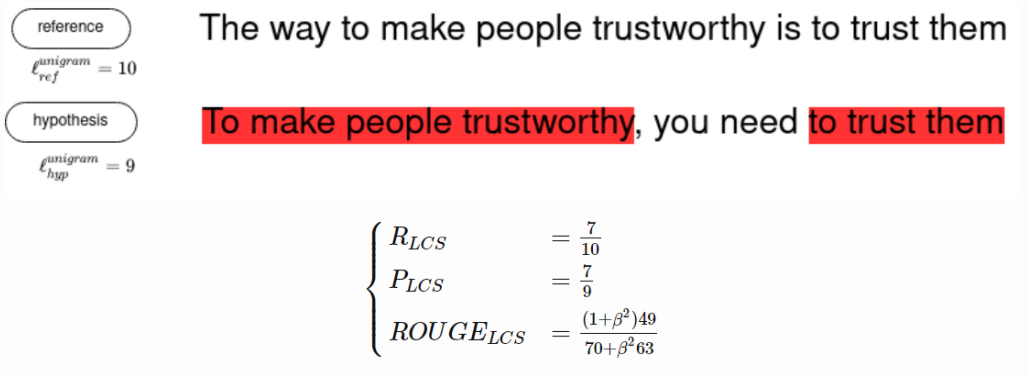


Figure 5.11: Example of calculating the Rouge-L score for given reference and candidate text [10].

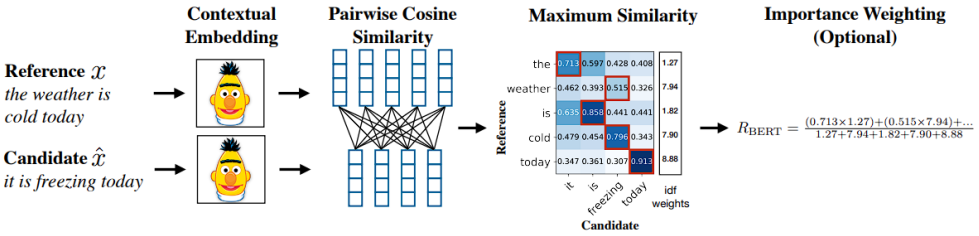


Figure 5.12: Example of calculating the Bert score for given reference and candidate text [44]. The token embeddings are usually represented as a vector, but the authors decided to use an image of the beloved kid show character Bert from Sesame Street.

generated code has the same functionality as the original one. However, this process is very complex and requires additional research.

As for the score of our metric, we propose the normalized metric from 0–1 based on the number of non-empty lines. It is essentially a ratio of faulty lines to the total number of non-empty lines in the generated code. The number of faulty lines will be taken from the compiler's error message, and the calculation will also depend on the type of error. For example, the missing variable/constant error is less severe than the syntax error. The algorithm then takes the most severe error found on the given line and adds its weight to the total error sum. We can represent this calculation by the equation:

If we normalize the weights of errors, we will also get a normalized final result where the lower the value, the better the result. When evaluating models, we will be using this metric's inverted value. This means that the closer the value is to 1, the better. This approach is more sensible and easier to comprehend for the reader.

Because this metric is parametrized by choosing the values of the errors, the results will vary according to the user's preferences. We decided to evaluate four types of errors with given values:

- Missing identifier (0.2),
- Already defined (0.3),
- Wrong value (according to variables type) (0.4),

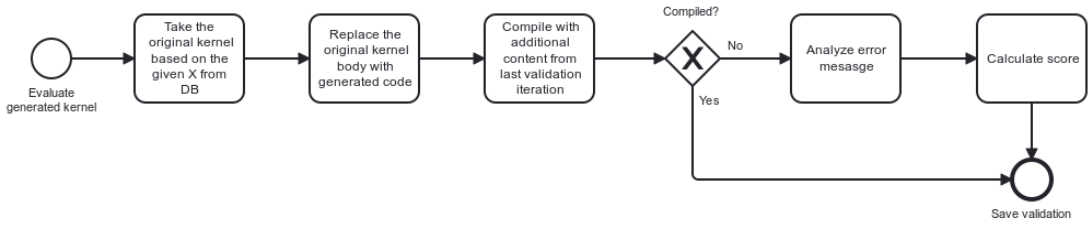


Figure 5.13: The evaluation process of the new compile metric for the generated kernels.

$$S = \frac{\sum_{i=0}^{n} \max(\mathbf{e_i})}{n}$$

S	Score of the kernel.
i	Line index.
n	Number of lines.
$\mathbf{e_i}$	Weights of errors on i-th line.

Figure 5.14: Equation of our evaluation metric.

- Missing type (definition without the type (0.6),
- Syntax error (1.0).

Again, these values are subjective and do not necessarily reflect the reader's opinion. An important disclaimer for this metric is that the implemented version of this metric uses regexes to analyze the error messages from the compiler; however, the error messages may differ based on the version of the used compiler. The information about the used compiler is in the dataset stored in the validation of every kernel.

5.4 Baseline

As a baseline, we trained the classical Encoder-Decoder Transformer model from scratch exclusively on our dataset. The used model contains four encoder layers and three decoder layers, and a token representation vector size was set to 400 with four Attention heads. We used variant dropout value where for the first 20 epochs, we set dropout to 0.4 and then gradually decreased the dropout down to 0.1 for the last 40th epoch. We used a tokenizer for GPT-2 with a size of 50,258 tokens. Our model summarizes almost $\sim$ 49M parameters. After the first 20 epochs, we also tried to train the model using teacher-forcing and the auto-regressive mode to ensure better generalization and further decrease the loss value. This, however, does not seem to do any effect on the models' performance.

From the results 5.4, despite the decent stable loss decrease, we see that the model was not able to adapt well to the dataset. This may have several reasons. The model might be too small and was not able to fit the corpus well enough. This would also indicate a relatively large loss value even for the last epoch. Another reason might be an implementation fault or wrong weights initialization for the model. Also, another reason for this bad performance might be that the transformer models are few-shot learners, as we already mentioned. This means that these models work better when pretrained on multitask datasets to learn the very grasp of human language and then finetune these models on the specific downstream

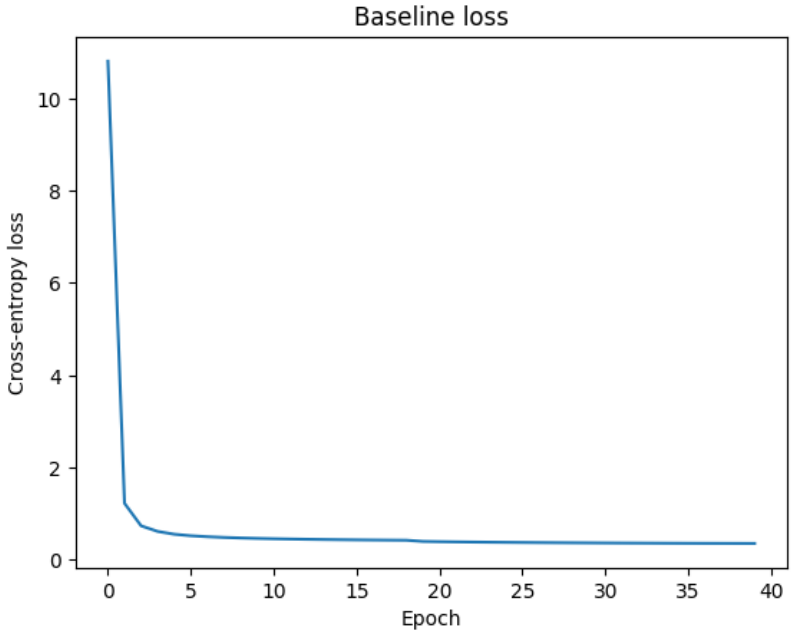


Figure 5.15: Baseline recorded loss value throughout the training process.

Kernel type	Bleu	Rouge1 prec.	Rouge2 prec.	Bert f1	Compile metric
___global___	0.058	0.012	0.002	0.03	0.044
___device___	0.07	0.014	0.002	0.042	0.032
___host___	0.045	0.012	0.002	0.044	0.033
total	0.063	0.013	0.002	0.037	0.037

Table 5.4: Baseline model evaluation grouped by kernel types using multiple evaluation metrics.

tasks. Also, the use of a tokenizer for GPT2 could worsen the performance as this tokenizer is finetuned for different model architectures.

An interesting fact is that the BERT score is deficient for all the kernel types. This means that even when the model predicted the token, it was not even remotely close to the one in ground truth. This, of course, can be caused by the fact that if the model predicted different tokens at the start of the decoded sentence, it could lead the model to a different context and very different output.

The baseline outputs examples are in the appendix B. The results are very distorted and not very useful in real-time coding assistance.

The baseline did not show outstanding results, but this does not necessarily means that our corpus is useless. The training process still shows a stable loss decrease; therefore, there are definitely features to learn. We now move to the general language models that have been pretrained on large general-purpose datasets.

5.5 Generally-pretrained Models

One of the motivations behind this report is to show that our new hand-crafted dataset can be used to finetune the general language model to write sensible code in CUDA. We decided to use several models that differ in size and architecture to see the impact and usefulness of our hand-crafted corpus and cross-validate the dataset performance. These models have the advantage compared to our baseline model in that they have already been pretrained on general-purpose corpora and somehow understand the human language. We finetuned these models using similar hyperparameters and the same dataset to see the differences between the models and their capabilities to produce sensible code.

We used the whole dataset for our finetuning. We train our models even on the non-CUDA kernels because they still represent correct C/C++ code, and the model can learn from them. As noted in this thesis, the CUDA language is more of an extension of the C++ code; therefore, we can use these samples for more data. We used weight decay set to $1e^{-2}$, dropout to 0.1, and learning rate scheduler with linear decrease, three epochs set as warm-up, and $1e^{-4}$ learning rate at peak. However, we trained these models for different times and numbers of epochs and batch sizes due to the models' limited computational resources and sizes.

5.5.1 T5

As a first generally pretrained model, we used Google's T5 (Text-to-Text Transfer Transformer) Encoder-Decoder transformer model [37] first introduced in 2019.

The T5 model was created and trained to perform a wide range of NLP tasks using a single architecture and a single set of weights. This is achieved through a process called „text-to-text transfer," where the input and output of the model are always in the form of text, regardless of the specific task being performed.

We use the version and checkpoint saved on the Hugging face website[6]. The model uses a hidden size of 512, with six decoder layers and eight Attention heads. This smaller version of the T5 has only a bit over 32,000 vocabulary size, which means that some special characters are not represented in the model. This configuration puts our T5 model on $\sim$ 60M parameters in the model.

This model was pretrained on C4 (Colossal Clean Crawled Corpus) introduced in the same paper as the T5 model [37]. C4 corpus is a large and diverse collection of text data that has been widely used to train LM models, particularly language models. The current version of the C4 corpus, released in 2021, contains over 750GB of text data in multiple languages and formats, including news articles, websites, scientific papers, and more.

According to the original paper [37], the T5 model has been trained to implement a language framework for general purposes. The figure from the original paper represents the use of this model:

To mimic this use case, we modified our data sampler to add the *„supplement code:"* prefix before the CUDA kernel.

We used a learning rate of $1e^{-4}$ at peak with a linear scheduler with warmup steps set to 3 epochs and a total number of epochs set to 30. Rest of the parameters were set as: $\beta = (0.9, 0.999)$, $\epsilon = 1e^{-6}$, and weight decay to $5e^{-3}$.

After 19 epochs, we evaluated mentioned metrics, including the new one. These are the results.

[6] T5-small model.

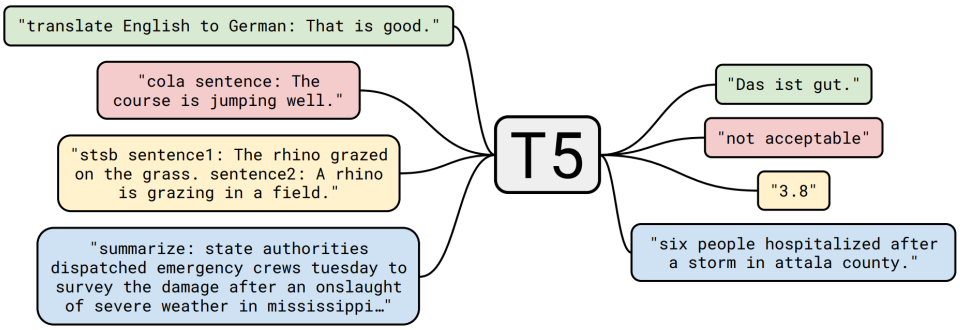


Figure 5.16: Examples of T5 model use-case [37].

```
supplement code:
// Method for operator/
// 1. param. constvec3& v1,
// 2. param. constvec3& v2,
// returns inline vec3
__host__ __device__
inline vec3 operator/ (const vec3& v1, const vec3& v2)
{
return vec3(v1[0] / v2[0],
            v1[1] / v2[1],
            v1[2] / v2[2]);
}
```

Figure 5.17: Example of training sample of our corpus. Note that this sample has a machine-generated comment.

Kernel type	Bleu	Rouge1 prec.	Rouge2 prec.	Bert f1	Compile metric
___global___	0.127	0.038	0.007	0.753	0.912
___device___	0.210	0.228	0.032	0.594	0.940
___host___	0.160	0.362	0.044	0.636	0.913
total	0.175	0.208	0.028	0.647	0.925

Table 5.5: T5 model evaluation grouped by kernel types using multiple evaluation metrics.

From the results table 5.5, we can see different results for different types of CUDA kernels. These results strongly correlate, but they do not precisely match as these types of kernels differ in their functionality. However, we can see a huge gap between rouge1 scores for *___global___* and other types of kernels. This indicates that model learned more about how to write per-thread kernels, which do not require calculating thread id and checking if the thread is not out of bounds for a given task. This confirms even the BLEU score, which is lower for the "global" kernels compared to the rest. This can be caused by two possible reasons. First, the "global" kernels make roughly 35% of the dataset and therefore make a

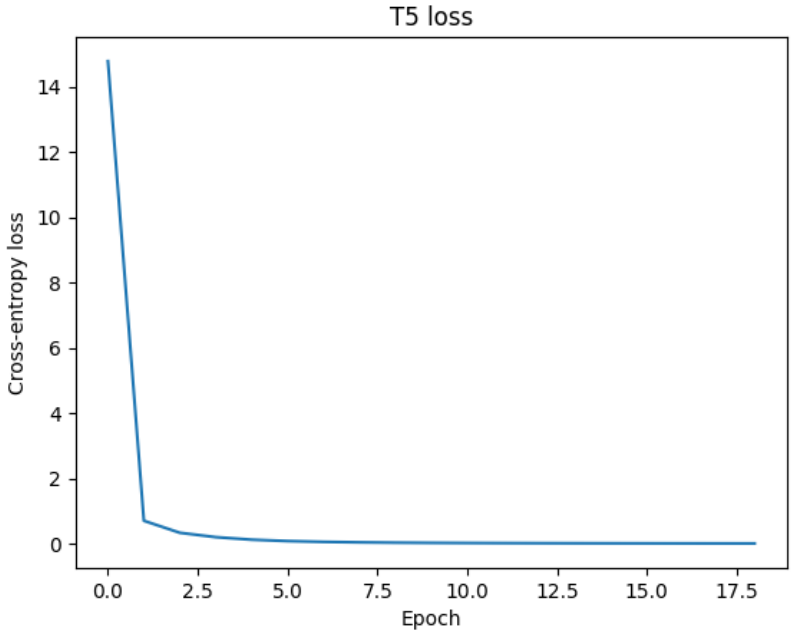


Figure 5.18: Recorded average loss value per epoch during the T5 model training process. The process ran for 19 epochs and achieved minimum in the 18th epoch with ~ 0.008 avg.

minority of the dataset, and secondly, the "device" and "host" kernels are syntactically very similar compared to the "global" kernels. This could be a reason for a stronger correlation between "device" and "host" kernels compared to the "global" ones.

Looking at the BERT score, we see a relatively strong correlation between ground truth and model prediction. We used the first layer (Embedding) to represent our tokens. This is a good result as it indicates that even though the predicted tokens were not the right ones, there had a similar meaning in the embedding layer of the model.

The last metric is our hand-crafted one, indicating how well the model generates compilationable code. We used weights of the errors described in the section about the used metrics 5.3.1. The data indicates that the model is quite able to generate syntactically correct code; however, there could be a problem with the T5 tokenizer. The T5 model uses smaller ($\sim$ 30K tokens) vocabulary that does not work with tabs and newline characters. We tried to compensate for this by adding these characters in post-processing, but there were still cases where for example, a line comment could absorb a line of code. This is undoubtedly preventable, but it requires deeper analyses of the generated code in post-processing.

Besides the result graphs, the crucial thing we must evaluate to understand how the model behaves is generated text. In appendix C, we put some examples of tasks that the model was asked to solve. We want to demonstrate how our baseline model behaves and analyze where there is room for improvement.

5.5.2 BART

Next on our list of generally pretrained models, we use BART [26], another Encoder-Decoder transformer, and the largest model we will be using in this thesis with the size of 400M parameters. We use checkpoint from the Hugging face website[7] with twelve Encoder and

[7]BART checkpoint.

Decoder layers, hidden size set to 1,024, and sixteen Attention heads. The vocabulary size of this model is 50,265 tokens which covers most of the cases. Another interesting fact about the BART model is that it uses GELU [19] as the activation function.

BART is a denoising Encoder-Decoder denoising auto-regressive model for general purposes published in 2019 by Mike Lewis and his team. BART combines the BERT and GPT architectures, using a bidirectional Encoder as BERT and a left-to-right Decoder as GPT.

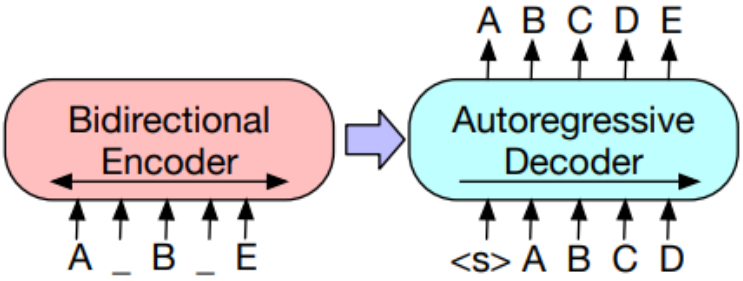


Figure 5.19: Representation of BART architecture as the combination of BERT encoder and GPT decoder [26]

Like the T5 model, BART can be used for general use, such as translation, summarization, and filling masked tokens (MLM), but unlike the T5 model, the BART does not use keywords before the input sentence to define the performed task. We decided to finetune the BART model using MLM, as the authors in the original paper describe, that the *<mask>* token masks spans of text and not only a single token. BART as a denoising model has been trained by inputting a shuffled and masked input sentence into the encoder and using the decoder to recover the original sentence state. This process, corresponding to the original paper [26], should work as a more general approach to model learning and ensure better generalization.

Kernel type	Bleu	Rouge1 prec.	Rouge2 prec.	Bert f1	Compile metric
___global___	0.115	0.083	0.029	0.708	0.343
___device___	0.092	0.181	0.085	0.588	0.355
___host___	0.101	0.201	0.086	0.562	0.322
total	0.101	0.148	0.065	0.629	0.349

Table 5.6: BART model evaluation grouped by kernel types using multiple evaluation metrics.

From the results on the BART model 5.6, we can observe that the model somehow was able to finetune to the downstream task. However, the achieved data are not outstanding, considering the fact that the BART model is the largest model we use in this report. However, upon further inspection, we can see quite high values for the BERT score in the same table, with the values achieving over 0.7 for the *___global___* category. Combining this information with values at the recorder loss figure 5.20, where the minimum average loss value in the last epoch is less than 0.05, we can argue that this is the case of model overfitting. The BERT score indicates that despite very low Rouge and Bleu scores, the model could predict very similar tokens compared to the ground truth. The very low training loss value shows that the model was able to learn the training set almost flawlessly.

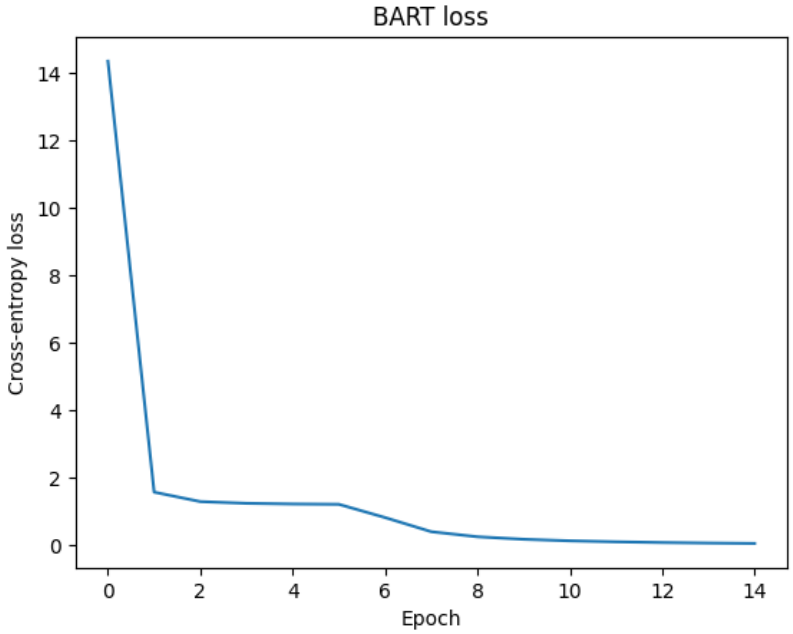


Figure 5.20: Recorded loss value throughout the training process. The highest value in the first epoch of 14.34. The lowest value in the last epoch of 0.0418.

Please note that we set the model's dropout values between 0.1 to 0.4 depending on the layer's type. We also used weight decay and shuffled the training corpus on epoch bases. Still, the model was able to overfit and worsen the generalization performance.

However, this discovery arguably also has a positive aspect. Given the fact that the models BART and also T5 were able to overfit on our corpus means that the corpus has to fulfill a certain quality of the dataset, and its data have to be somewhat clean. However, this is up for discussion.

5.5.3 GPT-2

As for our last general use pretrained model, we shifted our focus from Encoder-Decoder architectures and decided to finetune famous architecture GPT-2 [36]. This model was introduced in 2019 by Alec Radford and his team. The model follows decoder-only transformer architecture with the main task of predicting the next token in the sentence within a particular context.

We utilized a smaller version of the original model with around 80 million parameters and 12 layers [38], as the original one had 1.5 billion parameters and 48 decoder layers, which was limited by time and memory. We also alter the used dataset to use only the *___global___* kernels for training and evaluation. The motivation for this step is that the "global" kernels are arguably the most important part of this corpus. The programmer (language model) has to deal with thread identification, synchronization, and overall massive parallelism in these functions. We wanted to validate this part of the corpus and see that the model is able to learn only from these kernels. We also made a minor adjustment to the yielded kernels from our corpus to match the standard use of the GPT model, following the same approach as for other models. The GPT model was pretrained through multi-task training and designed to comprehend the task from the input prompt without the need for specific keywords.

We evaluated our model on multiple epochs of the narrowed dataset. We finetuned this model for 20 epochs, keeping a record of the loss value throughout the process, with a peak learning rate of $1e^{-4}$.

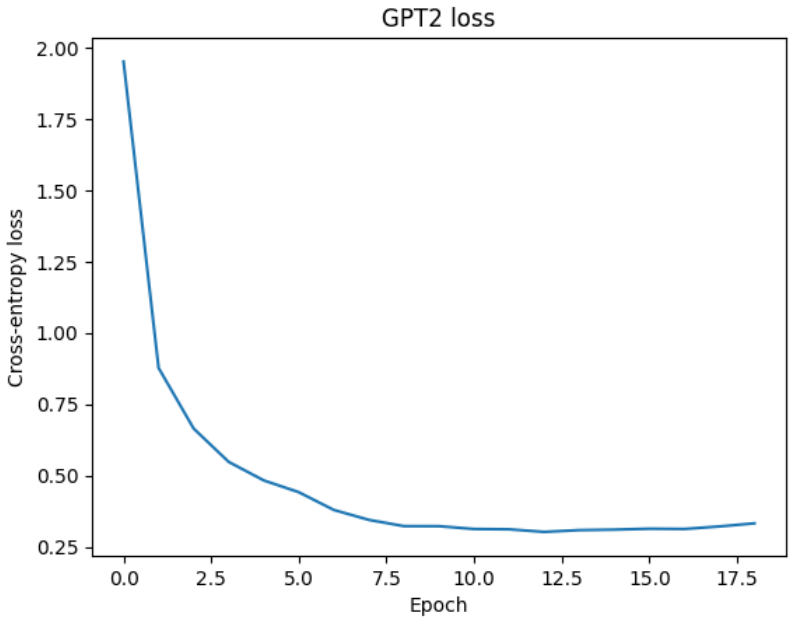


Figure 5.21: Recorded loss value throughout the training process for pretrained GPT2 model. The maximum value at the first epoch was 1.95 and the lowest at the 12th epoch with 0.302. After the 12th epoch, the loss started to stagnate or even bit increase.

From the recorded average loss value in figure 5.21, we can see the stable decrease down to 0.302 on the 12th epoch. The minimum loss is a bit higher than the previous generally pretrained models we may assume the difference might be caused by using narrowed corpus version but also by using different model architecture.

Kernel type	Bleu	Rouge1 prec.	Rouge2 prec.	Bert f1	Compile metric
___global___	0.167	0.121	0.056	0.557	0.842
___device___	0.202	0.134	0.08	0.592	0.888
___host___	0.155	0.111	0.071	0.499	0.812
total	0.187	0.128	0.071	0.575	0.868

Table 5.7: GPT2 model evaluation grouped by kernel types using multiple evaluation metrics.

The GPT2 model was able to achieve the best result from the generally-pretrained models and decent results overall, with the bleu value exceeding 0.2 for the *___device___* kernels. The rest of the metrics confirm this with relatively high values. The BERT score indicates that the model often kept the prediction at a similar representation level as the target. This also confirms the correlation between prediction and target size present in the appendix G.1d with the value of 0.53. the compile metric indicates mostly valid generated code from the GPT2 side.

For reference, we have included a few examples of the code generation samples from GPT2 in Appendix E.

5.6 Code-specialized Model

After training our baseline model and cross-checking the usability of our dataset on multiple general-purpose models, we decided that we should also compare these results with a model that has been trained specially for coding purposes. The idea is to compare the results of the specialized model to the rest of the pack and see if the coding model is able to learn more quickly and better the CUDA language and outperform even the scratch-trained baseline model.

5.6.1 CodeGen

The CodeGen is a Decoder-only left-to-right transformer architecture neural network model for code generation tasks. It was introduced in a research paper in 2021 by a team of researchers from Facebook AI [32]. CodeGen is designed to generate code from natural language descriptions of programming tasks.

We use checkpoint on Hugging face with 350M parameters[8].

This model has three versions; one is used as the starting point for another. We decided to use the middle "multi" version as, according to the authors, this version has been pre-trained on multiple languages. The next and last version, "mono", is the "multi" version finetuned further on Python code. The current model we are utilizing comprises 10 Encoder and 10 Decoder layers. The hidden size is set to 2,048, while the embedding size is set to 1,024. It's equipped with 16 Attention heads and a vocabulary size of 51,200. Like BART, CodeGen also employs the GELU activation function instead of the conventional ReLU.

Kernel type	Bleu	Rouge1 prec.	Rouge2 prec.	Bert f1	Compile metric
___global___	0.41	0.356	0.152	0.67	0.901
___device___	0.415	0.376	0.166	0.782	0.877
___host___	0.356	0.299	0.102	0.631	0.677
total	0.410	0.365	0.158	0.735	0.875

Table 5.8: CodeGen model evaluation grouped by kernel types using multiple evaluation metrics.

From the evaluated metrics in table 5.8, we see that the CodeGen model was able to adapt to our corpus well. The BLEU score over 0.4 on the evaluation dataset indicates a good understanding of the performed task; the rest of the metrics confirm this. We can see a slight hiccup in the ___*host*___ kernels where the evaluated values are below the average. One of the reasons why this is happening can be the small amount of these kernels in the dataset. Besides this, the model shows good performance even for the kernels with non-generated comments. This means that the model was able to focus on the important features in the input prompt despite the monotonous form of the comments.

The BERT score f1 value is an interesting result as it shows not-so-high precision between the target and prediction tokens. Combined with the high BLEU value, this could

[8]CodeGen model.

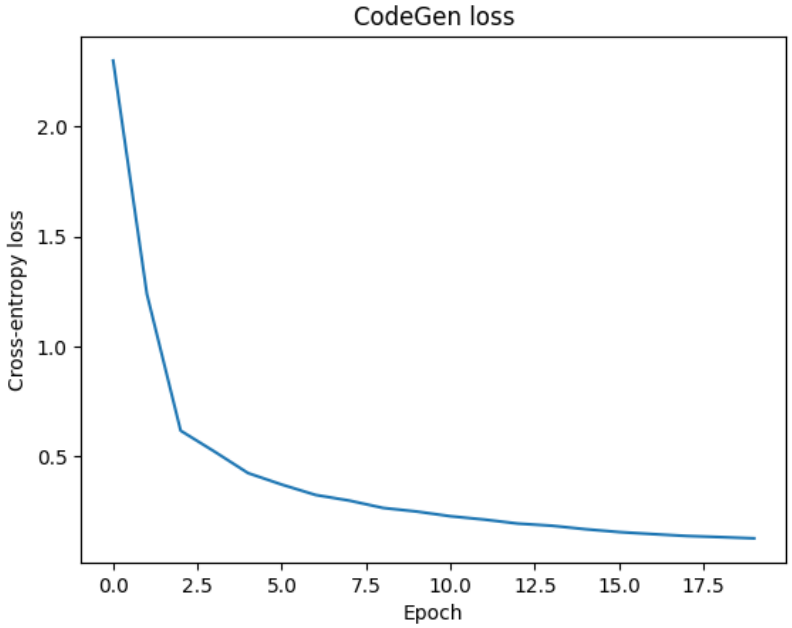


Figure 5.22: CodeGens recorded loss value throughout the training process. The maximum value of 2.3, Minimum in last epoch of 0.127 average per epoch.

mean that in cases where the model did not match the target sentence precisely, it often stranded in a completely different result. This also confirms one of the examples in the appendix F.3 where the prediction shows valid code but very different compared to the target.

The Compile metric shows that the model is able to generate mostly valid code with some slight problems with the ___*host*___ kernels.

The examples of the CodeGen code generation are available in appendix F, as well as the rest of the analysis.

5.7 Errors Analysis

After training and evaluating the models, we should look at the evaluated results in greater detail and analyze some of the most common errors we got. We looked at some of the data samples with the highest loss value and, therefore, those that make the biggest challenge for our models. We analyzed the differences and important aspects from the prompt input by which the models decide what code to generate.

While analyzing one of our models, we decided to sort the samples from the evaluation dataset based on their loss value. We then focused on the samples that had the highest criterion and identified several common aspects that could be the reason for their high error rate and poor results. We observed that most of these bad samples had not-generated comments, often taken out of context, irrelevant, or in a different language. It was evident that only 20% of the dataset had original not-generated comments. While these comments are typically richer and more authentic than machine-generated ones, they are also noisier. Our findings suggest that the function logic for generating kernel comments needs to be revised. The current logic is too machine-like and needs to capture the human commenting

style adequately. We also found quite often problems with character encoding. These kernels were often written using other writing styles than the alphabet, and this, using UTF-8 encoding, resulted in nonsense.

- // Sobre tablero se borra aquellas casillas indicadas por mask y se pone lo que haya en decisions
  // Debido a cómo se generan mask y decisions nunca coinciden con un valor en la misma posicion
- // complex math functions
- // gz e by,bz > 1 -> (1,1,32)(1,32,32)

Figure 5.23: Examples of bad comments. The first comment appears to be in Spanish, which is the language that the models weren't pretrained nor finetuned. The second comment is vague and does not provide the necessary information to generate valid code. The last comment seems to be taken out of broader context and does not say much useful.

We also looked at the cross-attention values of our models to see which input tokens the model pays the most attention to when generating the output sequence. These, of course, differ not only from prompt to prompt but also by the model. An interesting discovery was when looking at the T5 model; we discovered that the most „attention-worthy" tokens often were spaces. We demonstrate this in the example 5.24 with the according input. From this example, we can see that the highest values form columns and tend to belong to the space tokens. The reason for this is up to speculation. One of the reasons can be the tokenization of the data samples. Suppose the tokenizer is splitting the sentence into tokens mostly on space bases. In that case, the model might learn that spaces are essential for separating words and assigning meaning to the input. Another reason can be the pretrained token embedding weights where the space character is represented in a way that makes it stand out from the model.

Another thing we can see from figure 5.24 is the very high value for the „v" token in the input and output sequence. This arguably indicates the model's attention to what variable it is supposed to return from the function.

Earlier in the report, we discussed the differences between ___*global*___ and other kernels. We analyzed these differences as well. We looked at the models' predictions on the evaluation set. We searched for the use of the variables **threadIdx**, **blockIdx**, and **blockDim** in the predicted body in one line. This use of variables indicates that the model understands that it needs to identify the thread within the function using this built-in CUDA structure. This identification is used primarily in the ___*global*___ kernels, as the other types are usually called per-thread. We calculated these results and put them into the table 5.9. From the table, we can see that the models understand this aspect from the training dataset well as the number of positives makes almost 90% of the tested samples.

We also looked at the models' precision in using local memory. We analyzed the target and predicted sentences of the models and searched for the use of ___*shared*___ or ___*constant*___ memory. We then put these results into the table 5.10. The use of local memory is heavily imbalanced in the dataset, and therefore we also calculated precision and recall for each model 5.10.

```
// Kernel for dividing by two
__device__ float divideByTwo(float v)
{
    return v /2;
}
```

Listing 5.1: Input prompt and model's generated body for the cross-attention map below.

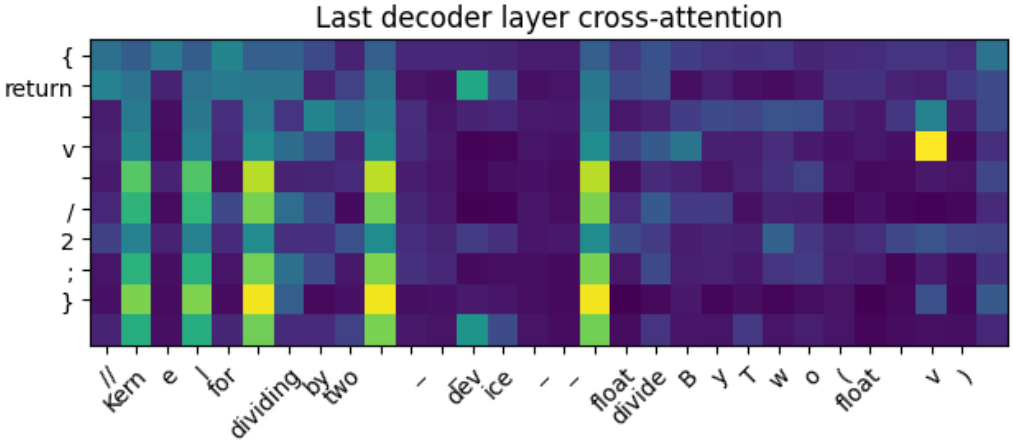


Figure 5.24: The cross-attention map summed up for all Attention heads in the last decoder layer of the T5 model.

	___global___ (positive)	other (positive)	___global___ (negative)	other (negative)
Baseline	0 (0.0%)	23,855 (100.0%)	14,992 (100,0%)	0 (0.0%)
T5	10,918 (72.49%)	22,973 (96.6%)	4,143 (27.51%)	808 (3.4%)
BART	869 (3.64%)	22,986 (96.36%)	14,647 (97.7%)	345 (2.3%)
GPT2	8.496 (63.3%)	24,742 (97.32%)	4,926 (36,7%)	682 (2.68%)
CodeGen	9,726 (64.58%)	23,367 (98.25%)	5,335 (35,42%)	415 (1.75%)

Table 5.9: Analyses of use of thread identification in kernel based on the kernel type. Positive means that the identification was used correctly, and negative means that the identification was either missing or unnecessary.

Another thing we were interested in is the correlation between the input & target sizes and the loss values. The logical implication is that the larger the input, the smaller the loss value on average, given that the input prompt is sensible and relevant information about the target code. On the other hand, a larger target sentence could indicate a higher loss value as the output gets more complicated and harder for the model to generate and gives more room for the model to skew its output to a completely different prediction compared to the target. The generated plots are put in the appendix. We evaluated this hypothesis for the used models and found that the correlation between the input prompt and the loss value is negligible and close to zero. The correlation between the target and predicted sentences is higher on average but is not too significant to draw conclusions from it 5.11.

	True positive	False positive	True negative	False negative	Precision	Recall
Baseline	0 (0.0%)	4 (0.01%)	35,026 (90.16%)	3,817 (9.83%)	0.00	0.00
T5	2,024 (5.21%)	2,195 (5.65%)	33,598 (80.54%)	3,340 (8.6%)	0.480	0.377
BART	567 (1.46%)	1,209 (3.11%)	33,628 (86.49%)	3,475 (8.94%)	0.319	0.14
GPT2	1,082 (2.79%)	1,724 (4.44%)	34,818 (89.63%)	1,222 (3.15%)	0.386	0.47
CodeGen	2,831 (7.29%)	729 (1.88%)	34,279 (88.25%)	1,004 (2.58%)	0.795	0.738

Table 5.10: Matching use of local memory comparing the target output to the predicted one. Positive means that the local memory was used in the target sentence. Negative means that the local memory was absent in the target code snippet. True and False means corresponding use of local memory in the predicted sentence.

	Baseline	T5	BART	GPT2	CodeGen
Input & loss corr.	0.02	0.13	0.12	0.08	0.18
Target & loss corr.	0.06	0.12	0.19	0.34	0.39

Table 5.11: Pearson correlation between the data sample's input and target and the loss value from the models' predictions. Calculated with the cross entropy loss function.

We also analyzed the lengths of the target and predicted outputs. We wanted to see if the models understood how big of an output they should generate from the input prompt. We generated graphs to visualize better the correlation between the target and predicted sentence lengths G. From these results, we can see that some models did not match the target size in most cases; however, the GPT2 and CodeGen models show a strong correlation between the two variables. The GPT2 and CodeGen models are the highest-performing models we used in this research according to the recorded metrics, and this correlation confirms it.

	Baseline	T5	BART	GPT2	CodeGen
Target & Prediction corr.	-0.14	0.48	0.02	0.53	0.78

Table 5.12: Pearson correlation between the data samples' target output and the prediction.

5.8 Results Summary & Discussion

In the last section, we discussed our analyses of the used models and their results. In this section, we would like to summarize these results and open the discussion shortly. The main idea behind this research is to create a helpful corpus that can be used for training language models to generate valid code in CUDA language. We argue that this task has been accomplished from the results of the used model. Some models performed better than others, and some did not converge to a usable state. Still, on the other part of the spectrum,

we have two models, GPT2 and CodeGen, that are able to generate sensible code and seem to understand the task from the input prompt.

	BLEU	Rouge1 prec.	Rouge2 prec.	BERT	Compile metric
Baseline	0.063	0.013	0.02	0.037	0.037
T5	0.175	0.208	0.028	0.647	0.925
BART	0.101	0.148	0.065	0.629	0.349
GPT2	0.187	0.128	0.071	0.575	0.868
CodeGen	0.410	0.365	0.158	0.735	0.875

Table 5.13: Summary of the models' performance on the evaluated metrics.

Upon further inspection from the user's view, we agree with the results and proclaim it as the best-performing model CodeGen, as it generates decent results even for vague comments and very specific use cases. We compared the CodeGen model to the recently very popular GPT4 [33], a general-purpose LLM with 1 trillion parameters. For comparison, the CodeGen model we used has 350 million parameters. GPT4 was trained for a much more extended period of time and on immensely larger data corpora. Still, after the comparison, we argue that our CodeGen model is close to the GPT4 performance in terms of generating CUDA code. The showcase of examples is in appendix H. Note that the GPT4, as the CodeGen, can generate different responses based on the used seed. We used seeds 1, 2, and 3 for the CodeGen model; however, we were unable to set seeds for the GPT4 through the API. This means that the results of the GPT4 may differ after repeating queries.

In all three examples, both models generated valid code. In the first example H.1, both models were asked to generate a matrix multiplication kernel. Both produced reasonable responses. Unfortunately, neither of them delivers optimized code but rather straightforward, easy solutions. In the second example H.2, the CodeGen produces a better solution as the GPT4's snippet did not cover bases where vectors are larger than the number of running threads. In the third and most challenging example H.3, the models were asked to produce a matrix multiplication kernel that is optimized using ___*shared*___ memory. Both generated very good solutions. The CodeGen's lack of using the structural brackets could be explained by pretraining on the Python data samples; however, the number of immersed code lines is always one, which makes the prediction valid. Unfortunately, the excessive use of "if" statements and unusual conditions decrease the overall quality.

Chapter 6

Conclusions and Possible Improvements

In this last part, we would like to conclude our work presented in this report. In this research, we combined two, at first sight, very far-fetched topics; Natural Language Processing and optimized computing on the GPU. We dedicated this research to finding out if the language models are able to learn unorthodox paradigms for which they are not usually trained. For this purpose, we choose the CUDA as an extension to the C/C++ language. In order to find out the Language models' capabilities of learning CUDA, we first had to create a corpus that we could use for finetuning these models. We swiftly introduced the reader to the world of NLP and machine learning. We described the current state-of-the-art transformer architecture, which is used not only in NLP but dominates the machine learning science field. We then proceeded to the description of the process of creating a new dataset and the necessary steps we took. We analyzed our newly acquired data, not only from the code point of view but also looked at the source reliability. Of course, we also discussed what the CUDA language is, how it differs from the mainstream languages, and what are the crucial features that the model should learn in order to generate the code properly.

After the introduction and the theoretical part of this work, we proposed the pipeline for obtaining and processing data. We performed extensive analyses of these newly acquired data not only from the perspective of the code but also from the reliability of the code's source. We proposed the design, training & evaluation process, and experiments to perform in order to validate the usability of our corpus and to see the selected models' performance. We selected models with different architectures and backgrounds to maximize the diversity of the "agents" which used our corpus. We evaluated and analyzed the results and drew conclusions from them. In the end, we also compared our best-performing model to the GPT4 model with 1 trillion parameters, surpassing the famous ChatGPT. We selected several examples on which it is apparent that our best-performing model is able to compete with the GPT4 model when it comes to CUDA code generation.

As an outlook for future research, we could look at some of the potential improvements that could help even the outstanding need to look at the performed Data Analyses section 5.1.3. We mentioned that almost 80% of the corpus samples have machine-generated comments. This lack of documentation from the source code is a huge problem and causes problems in the validation sets for the few samples with human-made documentation and, of course, more problems in real-world data as the primary use case for these models is

to generate valid code based on the programmers hints in the form of comments. Our corpus is trying to mimic the doxygen form of method commentary, which is arguably the most common form of documenting code; however, from the results, we can see that it lacks variety and some form of data augmentation. One of the possible improvements to the corpus would be to implement a comment augmentation class that would generate and modify the comments for data samples without one. This process would help create richer and more human-like comments that would force the trained models to learn more critical features and generalize. The specific details of this process are the subjects of the potential research. For other potential improvements, we can look at the described section about competitive programming 3.5 for other potential improvements. Here we describe the process of creating a code generator that can compete with human programmers.

In conclusion, at the beginning of this research, we set up a goal to explore the capabilities of the LLMs and teach them how to generate sensible CUDA code. Also, if possible, to create a model that would serve as an example of language models' capabilities of learning even lesser-known and used paradigms of the coding world. From the results published in this report, we argue that these goals were met and fulfilled. The material results of this thesis are a corpus containing roughly 500,000 CUDA kernels, commented, analyzed, and ready to use, and the finetuned version of the CodeGen model, which, arguably, produces valid and sensible results and meets the users' expectations.

Bibliography

[1] *Encoder-decoder Seq2Seq for machine translation.* Available at: https://d2l.ai/chapter_recurrent-modern/seq2seq.html.

[2] *Gradient descent in machine learning.* Available at: https://www.javatpoint.com/gradient-descent-in-machine-learning.

[3] *Understanding GPU Architecture.* Available at: https://cvw.cac.cornell.edu/GPUarch/simt_warp.

[4] *King - man + woman = queen: The hidden algebraic structure of words.* Jul 1970. Available at: https://www.ed.ac.uk/informatics/news-events/stories/2019/king-man-woman-queen-the-hidden-algebraic-struct.

[5] *CUDA toolkit documentation.* Dec 2022. Available at: https://docs.nvidia.com/cuda/.

[6] Amidi, A. and Amidi, S. *Recurrent neural networks cheatsheet star.* Available at: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks.

[7] Amidi, A. and Amidi, S. *Recurrent neural networks cheatsheet star.* Available at: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks.

[8] Bjertnes, L., Tørring, J. O. and Elster, A. C. *LS-CAT: A Large-Scale CUDA AutoTuning Dataset.* 2021.

[9] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. et al. Language Models are Few-Shot Learners. *CoRR.* 2020, abs/2005.14165. Available at: https://arxiv.org/abs/2005.14165.

[10] Brutti Mairesse, C. *Rouge and BLEU scores for NLP model evaluation.* Dec 2021. Available at: https://clementbm.github.io/theory/2021/12/23/rouge-bleu-scores.html.

[11] Cho, K., Merrienboer, B. van, Gulcehre, C., Bahdanau, D., Bougares, F. et al. *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation.* 2014.

[12] Chowdhary, K. R. *Natural Language Processing.* New Delhi: Springer India, 2020. 603–649 p. ISBN 978-81-322-3972-7. Available at: https://doi.org/10.1007/978-81-322-3972-7_19.

[13] Chung, K. L. Markov chains. *Springer-Verlag, New York*. Springer. 1967.

[14] Dabre, R. and Fujita, A. Softmax Tempering for Training Neural Machine Translation Models. *ArXiv preprint arXiv:2009.09372*. 2020.

[15] Goodfellow, I., Bengio, Y. and Courville, A. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[16] Gurney, K. *An Introduction to Neural Networks*. London: [b.n.], 2017. ISBN 9781315273570. Available at: https://www.taylorfrancis.com/books/mono/10.1201/9781315273570/introduction-neural-networks-kevin-gurney.

[17] Hayati, S. A., Olivier, R., Avvaru, P., Yin, P., Tomasic, A. et al. Retrieval-based neural code generation. *ArXiv preprint arXiv:1808.10025*. 2018.

[18] He, T., Zhang, J., Zhou, Z. and Glass, J. *Exposure Bias versus Self-Recovery: Are Distortions Really Incremental for Autoregressive Text Generation?* 2021.

[19] Hendrycks, D. and Gimpel, K. Bridging Nonlinearities and Stochastic Regularizers with Gaussian Error Linear Units. *CoRR*. 2016, abs/1606.08415. Available at: http://arxiv.org/abs/1606.08415.

[20] Hiemstra, D. and de Vries, A. *Relating the new language models of information retrieval to the traditional retrieval models*. Netherlands: University of Twente, june 2000. CTIT Technical report series. Imported from CTIT.

[21] Hochreiter, S. and Schmidhuber, J. Long Short-Term Memory. *Neural Computation*. november 1997, vol. 9, no. 8, p. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735. ISSN 0899-7667. Available at: https://doi.org/10.1162/neco.1997.9.8.1735.

[22] Horan, C. *Tokenizers: How machines read*. FloydHub Blog, Feb 2020. Available at: https://blog.floydhub.com/tokenization-nlp/.

[23] Kirk, D. B. and Wen Mei, W. H. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[24] Kudo, T. Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates. In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Melbourne, Australia: Association for Computational Linguistics, July 2018, p. 66–75. DOI: 10.18653/v1/P18-1007. Available at: https://aclanthology.org/P18-1007.

[25] Lamb, A., Goyal, A., Zhang, Y., Zhang, S., Courville, A. et al. *Professor Forcing: A New Algorithm for Training Recurrent Networks*. 2016.

[26] Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A. et al. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. *CoRR*. 2019, abs/1910.13461. Available at: http://arxiv.org/abs/1910.13461.

[27] Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J. et al. Competition-level code generation with alphacode. *Science*. American Association for the Advancement of Science. 2022, vol. 378, no. 6624, p. 1092–1097.

[28] Lin, C.-Y. Rouge: A package for automatic evaluation of summaries. In: *Text summarization branches out*. 2004, p. 74–81.

[29] Medsker, L. R. and Jain, L. Recurrent neural networks. *Design and Applications*. 2001, vol. 5, p. 64–67.

[30] Mielke, S. J., Alyafeai, Z., Salesky, E., Raffel, C., Dey, M. et al. Between words and characters: A Brief History of Open-Vocabulary Modeling and Tokenization in NLP. *CoRR*. 2021, abs/2112.10508. Available at: https://arxiv.org/abs/2112.10508.

[31] Mikolov, T., Chen, K., Corrado, G. and Dean, J. Efficient estimation of word representations in vector space. *ArXiv preprint arXiv:1301.3781*. 2013.

[32] Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H. et al. *CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis*. 2023.

[33] OpenAI. *GPT-4 Technical Report*. 2023.

[34] Papineni, K., Roukos, S., Ward, T. and Zhu, W.-J. Bleu: a method for automatic evaluation of machine translation. In: *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 2002, p. 311–318.

[35] Pramoditha, R. *The concept of artificial neurons (perceptrons) in neural networks*. Towards Data Science, Dec 2021. Available at: https://towardsdatascience.com/the-concept-of-artificial-neurons-perceptrons-in-neural-networks-fab22249cbfc.

[36] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D. et al. Language models are unsupervised multitask learners. *OpenAI blog*. 2019, vol. 1, no. 8, p. 9.

[37] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S. et al. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *CoRR*. 2019, abs/1910.10683. Available at: http://arxiv.org/abs/1910.10683.

[38] Sanh, V., Debut, L., Chaumond, J. and Wolf, T. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. In: *NeurIPS EMC^{2}Workshop*. 2019.

[39] Song, F. and Croft, W. B. A General Language Model for Information Retrieval. In: *Proceedings of the Eighth International Conference on Information and Knowledge Management*. New York, NY, USA: Association for Computing Machinery, 1999, p. 316–321. CIKM '99. DOI: 10.1145/319950.320022. ISBN 1581131461. Available at: https://doi.org/10.1145/319950.320022.

[40] Tamura, Y. *Multi-head attention mechanism: „queries", „keys", and „values," over and over again*. Yasuto Tamura https://www.data-science-blog.com/wp-content/uploads/2016/09/data-science-blog-logo.png, May 2022. Available at: https://data-science-blog.com/blog/2021/04/07/multi-head-attention-mechanism/.

[41] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L. et al. Attention is All you Need. In: Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R. et al., ed. *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2017, vol. 30. Available at: https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

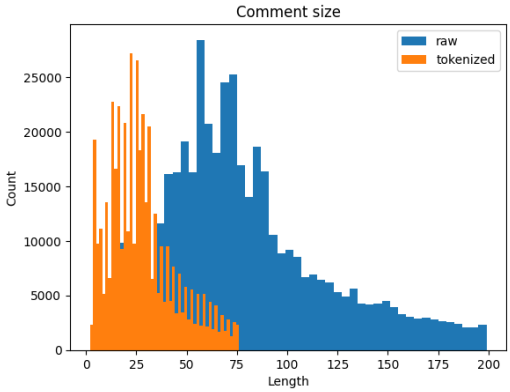
[42] Xu, Y. and Zhu, Y. A Survey on Pretrained Language Models for Neural Code Intelligence. *ArXiv preprint arXiv:2212.10079*. 2022.

[43] Yoo, J. Y., Morris, J. X., Lifland, E. and Qi, Y. Searching for a Search Method: Benchmarking Search Algorithms for Generating NLP Adversarial Examples. *CoRR*. 2020, abs/2009.06368. Available at: https://arxiv.org/abs/2009.06368.

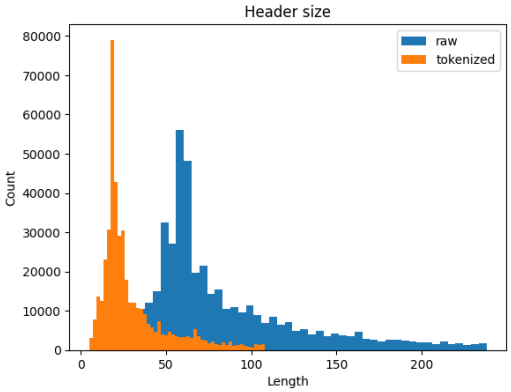
[44] Zhang, T., Kishore, V., Wu, F., Weinberger, K. Q. and Artzi, Y. BERTScore: Evaluating Text Generation with BERT. *CoRR*. 2019, abs/1904.09675. Available at: http://arxiv.org/abs/1904.09675.

Appendix A

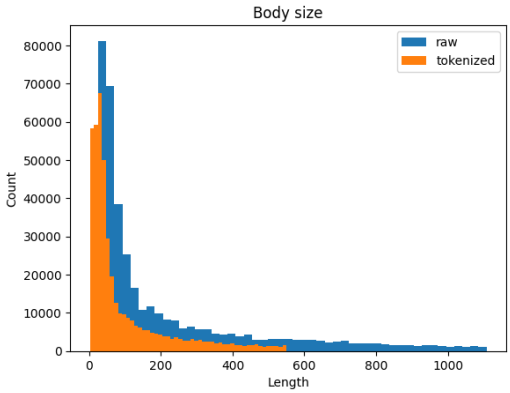
Corpus statistics



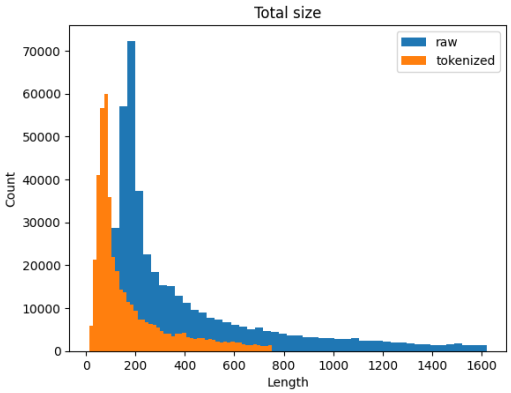
(a) Histogram of comment lengths character-wise and tokenized using GPT2 tokenizer.



(b) Histogram of header lengths character-wise and tokenized using GPT2 tokenizer.



(c) Histogram of kernels' body lengths character-wise and tokenized using GPT2 tokenizer.



(d) Histogram of whole kernels' lengths character-wise and tokenized using GPT2 tokenizer.

Figure A.1: Kernel length statistics in the obtained dataset. The plots use quantile of 0.9 to get rid of the far-fetched outliners.

Appendix B

Baseline code generating results

Examples of the models' performance	

Prompt	`// The squared distance between two vector,`
	`// only care xyz component`
	`__host__ __device__ __forceinline__`
	`float squared_distance(const float3& v3, const float3& v4)`

Target	`{`
	`  return (v3.x - v4.x) * (v3.x - v4.x) +`
	`  (v3.y - v4.y) * (v3.y - v4.y) +`
	`  (v3.z - v4.z) * (v3.z - v4.z);`
	`}`

Prediction	`________//////////////////////////////`

Table B.1: Baseline - prediction example n.1.

Examples of the models' performance	
Prompt	`// subtract` `inline __host__ __device__ int2 operator-(int2 a, int2 b)`

Target	`{`
	`  return make_int2(a.x - b.x, a.y - b.y);`
	`}`

Prediction	`{ return float a.x - b.x; }`

Table B.2: Baseline - prediction example n.2.

Appendix C

T5 code generating results

Examples of the models' performance	
Prompt	supplement code: // Subtraction __host__ __device__ __forceinline__ self_t operator-(difference_type n)

Target	{
	    return self_t(input1 - n, input2 - n, input3 - n, op);
	}

Prediction	{
	    return self_t(input1 - n, input2 - n, input3 - n, op);
	}

Table C.1: T5 model- example 1. Example of a simple override of the minus operator, which the model executed flawlessly. The loss value of this example was 0.03.

Examples of the models' performance

Prompt

```
suplement code:
// Function for adding vectors
// param1 float* v1
// param2 float* v2
// param3 float* out
// param4 int size
__global__ void addVectors(
    float* v1,
    float* v2,
    float* out,
    int size)
```

Target

```
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    for (;idx < size; idx += blockDim.x) {
        out[idx] = v1[idx] + v2[idx];
    }
}
```

Prediction

```
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    while (idx < size; idx+= blockDim.x) {
        out[idx] = v1[idx] + v2[idx];
    }
 }
```

Table C.2: T5 model - example 2. Example of simple element-wise addition of two vectors. From the prediction, we can see that the model figured out the executed task but struggled with the syntax as it got confused about whether to use "while" or "for" cycle. The loss value of this example was 0.933 using cross-entropy.

Appendix D

BART code generating results

Examples of the BART models' performance	

Prompt	`// value`
	`__global__ void VoteAllKernel2(unsigned int *input,`
	`  unsigned int *result,`
	`  int size)`

Target	`{`
	`  int tx = threadIdx.x;`
	
	`  int mask = 0xffffffff;`
	`  result[tx] = __all_sync(mask, input[tx]);`
	`}`

Prediction	`{`
	`  int tid = threadIdx.x + blockDim.x*blockIdxreads();`
	
	`  if (tid < size) return;`
	
	`  // stop in sharedizes!`
	`  mask = %0000[id + s, input //`
	`  if(tid == 1) {`
	`  return;`
	`  }`
	`  else {}`
	`}`

Table D.1: BART - code generation sample example. The model understood that the logic lies in masking the input parameter. However, it could not assemble the logic correctly.

Appendix E

GPT2 code generating results

1. Example of the GPT2 models' performance

Prompt

```
// brief Shift-left then add.
// Returns (\p x << \p shift) + \p addend.
__device__ __forceinline__ unsigned int SHL_ADD(
    unsigned int x,
    unsigned int shift,
    unsigned int addend)
```

Target

```
{
    unsigned int ret;
#if CUB_PTX_ARCH >= 200
    asm ("vshl.u32.u32.u32.clamp.add %0, %1, %2, %3;" :
        "=r"(ret) : "r"(x), "r"(shift), "r"(addend));
#else
    ret = (x << shift) + addend;
#endif
    return ret;
}
```

Prediction

```
{
    unsigned int ret;
    asm volatile("xor.u32.u32.u32.clamp.add %0, %1;"
        : "=r"(ret) );
    return ret;
}
```

Table E.1: GPT2 - code generation sample example. The model understood that the code snippet is written in an assembler. The target snippet uses vector-shift-left instruction, which the model did not use. Instead, it used xor, which slightly alters the code's logic.

2. Example of the GPT2 models' performance	

Prompt	`// Templated copy (base case)`
	
	`__device__ __forceinline__ void CopySegment(`
	`  T* out, ///< [out] Out array`
	`  T* in, ///< [in] Input array`
	`  Int2Type iteration)`

Target	`{}`

Prediction	`{`
	`  out[ITERED_MEMORY] = in[ITERED_MEMORY];`
	`  CopySegment(out, in, Int2Type<ITERED_MEMORY>());`
	`}`

Table E.2: GPT2 - code generation sample example. In this example, we can see that the GPT2 model generated code despite the empty target.

Appendix F

CodeGen code generating results

Examples of the CodeGen models' performance

Prompt

```
// brief Access operator
// param row index of row
// param col index of column
// return Reference to specified element
__host__ __device__ inline
float & operator()(unsigned char row, unsigned char col)
```

Target

```
{
    if (col == 0) return r[row].x;
    if (col == 1) return r[row].y;
    if (col == 2) return r[row].z;
    return r[0].x;
}
```

Prediction

```
{
      if (col == 3) return this->row(row).w;
      if (col == 2) return this->row(row).z;
      if (col == 1) return this->row(row).y;
      return this->row(row).x;
}
```

Table F.1: CodeGen - code generation sample example. It is apparent that the model was able to generate very accurate predictions. It that even understood that the kernel is part of a bigger structure and that there are variables that are not part of the context. The model tried to predict these variables and correctly accessed them by using the „this“ keyword.

Examples of the CodeGen models' performance	

Prompt	`// for(z)? Or some other order?`
	`__device__ int gridIndex3Dto1D(`
	`  int x,`
	`  int y,`
	`  int z,`
	`  int gridResolution)`

Target	`{`
	`  return x +`
	`  y * gridResolution +`
	`  z * gridResolution * gridResolution;`
	`}`

Prediction	`{`
	`  return x +`
	`  y * gridResolution +`
	`  z * gridResolution * gridResolution;`
	`}`

Table F.2: CodeGen - code generation sample example. The model was able to correctly calculate the index in the flattened 3D structure.

Examples of the CodeGen models' performance

Prompt

```
// Must be invoked with 256 threads,
// one for each output histogram element
__global__ void
histogramPerBlockFinalReduction(
    unsigned int *pHist,
    unsigned int cHist )
```

Target

```
{
    unsigned int sum = 0;
    for ( int i = 0; i < cHist; i++ ) {
        sum += tempHist[i][threadIdx.x];
    }
    pHist[threadIdx.x] = sum;
}
```

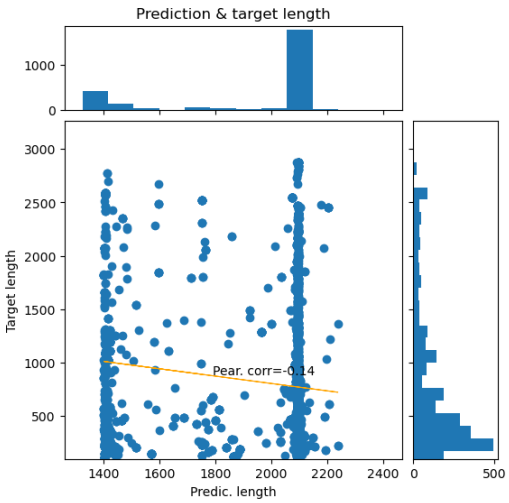
Prediction

```
{
  extern __shared__ unsigned int temp[];
  temp[threadIdx.x] = 0;
  __syncthreads();
  for (int i = threadIdx.x; i < 256; i += blockDim.x ) {
      int index = blockIdx.x * i + threadIdx.x;
      atomicAdd( &temp[index], cHist[ index ]);
  }
  __syncthreads();
  atomicAdd( &pHist[threadIdx.x], temp[threadIdx.x] );
}
```

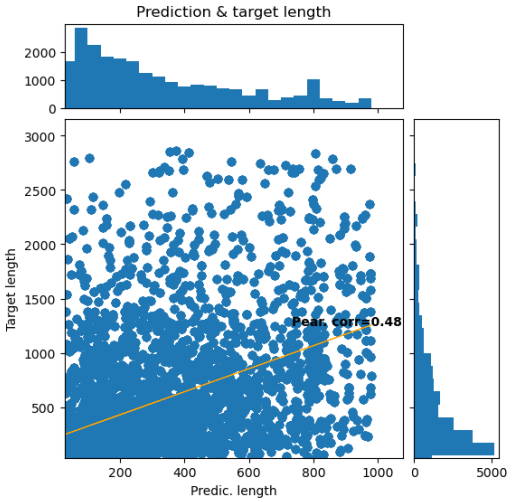
Table F.3: CodeGen - code generation sample example. The model went with a different route compared to the target snippet. It tried to generate code with the ___*shared*___ optimized memory but did not fill out the size of the array. The model used correct ___*syncthreads*___ primitives and also used **atomicAdd** to prevent the common conflicts when accessing shared memory from multiple threads. However, the addition inside the for loop uses an incorrect variable.

Appendix G

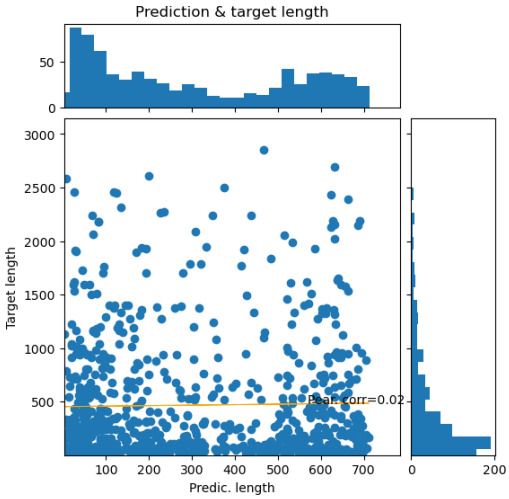
Target & prediction size comparison



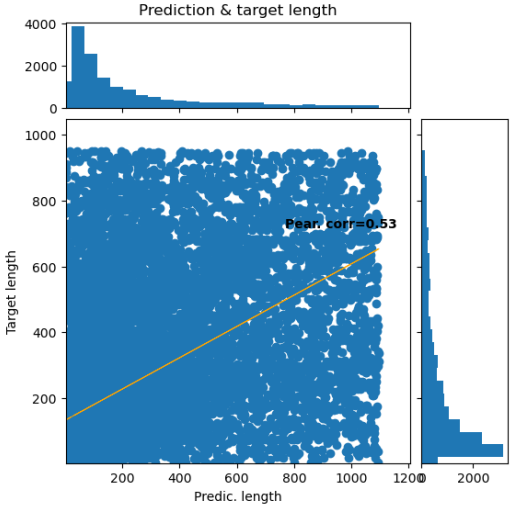
(a) Baseline model correlation between the target output and the prediction. The plot shows a slight negative correlation -0.13 and two main columns of prediction length.



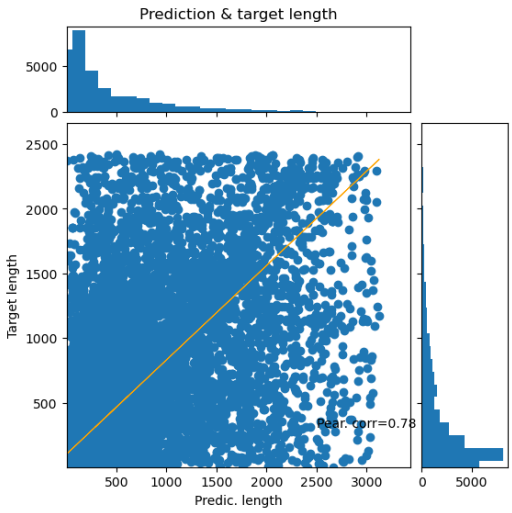
(b) T5 model correlation between the target output and the prediction. With the Person correlation of 0.48 T5 shows that it was able learn the basics of the kernels complexity from the input prompt.



(c) BART model correlation between the target output and the prediction. The Pearson correlation index indicates that the BART prediction sentence length does not depend at all on the target length.



(d) GPT2 model correlation between the target output and the prediction. The correlation between the target and the prediction length is significant with the value of Pearson correlation index of 0.53.



(e) CodeGen model correlation between the target and the prediction lengths. The solid 0.78 Pearson correlation indicates a good model's understanding of the input prompts and generating corresponding responses.

Figure G.1: Correlation of used models between target sentence length and prediction length.

Appendix H

Comparison with GPT4

Comparison between used CodeGen and GPT4

Prompt

```
// function for matrix multiplication
__global__ void matrixMul(
    float* A,
    float* B,
    float* out,
    int row_size,
    int col_size)
```

CodeGen

```
{
    int i = blockDim.y * blockIdx.y + threadIdx.y;
    int j = blockDim.x * blockIdx.x + threadIdx.x;
    float sum = 0.0;
    if (i < row_size && j < col_size) {
      for (int k = 0; k < col_size; k++) {
        sum += A[i * col_size + k] * B[k * col_size + j];
      }
      out[i * col_size + j] = sum;
    }
}
```

GPT4

```
{
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;

  if(row < row_size && col < col_size) {
    float sum = 0;

    for(int i = 0; i < row_size; i++) {
        sum += A[row * row_size + i] * B[i * col_size + col];
    }

    out[row * col_size + col] = sum;
  }
}
```

Table H.1: Comparison between used CodeGen and GPT4. The prompt requires code for matrix multiplication. Both models produced very similar code.

Comparison between used CodeGen and GPT4	
Prompt	`// function for adding two vectors` `__global__ void matrixAdd(float* v1, float* v2, int size)`

CodeGen	`{`
	`  for (int i = blockIdx.x * blockDim.x + threadIdx.x;`
	`  i < size;`
	`  i += blockDim.x * gridDim.x)`
	
	`  v1[i] += v2[i];`
	`}`

GPT4	`int i = blockIdx.x * blockDim.x + threadIdx.x;`
	`if (i < size) {`
	`  v1[i] += v2[i];`
	`}`

Table H.2: Comparison between used CodeGen and GPT4. The prompt requires code for Adding two vectors. CodeGen actually generated an identical version to the GPT4, but also generated this one, which is better because it covers the use case when the vector size is larger than the number of threads.

Comparison between used CodeGen and GPT4

Prompt

```
// function for optimized matrix multiplication
// using shared memory
__global__ void optimizedMatrixMul(
    float* A,
    float* B,
    float* out,
    int row_size,
    int col_size)
```

CodeGen

```
{
  //shared memory to store the kernel
  // that is needed to compute each element of the block
  __shared__ float sA[TILE_WIDTH][TILE_WIDTH];
  __shared__ float sB[TILE_WIDTH][TILE_WIDTH];
  int bx = blockIdx.x, by = blockIdx.y;
  int tx = threadIdx.x, ty = threadIdx.y;
  int row = by * blockDim.y + ty;
  int col = bx * blockDim.x + tx;
  float sum = 0;
  for(int t = 0; t < ceil(1.0 * blockDim.x/TILE_WIDTH); t++)
  {
      if(row < row_size && t*TILE_WIDTH+tx < col_size)
        sA[ty][tx] = A[row*col_size + t*TILE_WIDTH + tx];
      else
        sA[ty][tx] = 0;
      if (t*TILE_WIDTH+ty < col_size && col < col_size)
        sB[ty][tx] = B[(t*TILE_WIDTH+ty)*col_size + col];
      else
        sB[ty][tx] = 0;
      __syncthreads();
      for(int i = 0; i < TILE_WIDTH; i++) {
          sum += sA[ty][i] * sB[i][tx];
      }
      __syncthreads();
  }
  if(row < row_size && col < col_size)
    out[row*col_size + col] = sum;
}
```

Comparison between used CodeGen and GPT4

GPT4

```
{
  __shared__ float shared_A[TILE_SIZE][TILE_SIZE];
  __shared__ float shared_B[TILE_SIZE][TILE_SIZE];

  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int bx = blockIdx.x;
  int by = blockIdx.y;

  int row = by * TILE_SIZE + ty;
  int col = bx * TILE_SIZE + tx;

  float result = 0.0;

  for (int i = 0; i < (col_size-1) / TILE_SIZE + 1; i++)
  {
    if (row_size > (i * TILE_SIZE + tx) &&
        col_size > (i * TILE_SIZE + ty))
    {
      shared_A[ty][tx] = A[row_size *
                           (i * TILE_SIZE + ty) + col];
      shared_B[ty][tx] = B[col_size *
                           (row + i * TILE_SIZE) + tx];
    }
    else
    {
        shared_A[ty][tx] = 0.0;
        shared_B[ty][tx] = 0.0;
    }

    __syncthreads();

    for (int j = 0; j < TILE_SIZE; j++)
    {
        result += shared_A[ty][j] * shared_B[j][tx];
    }

    __syncthreads();
  }

  if (row < row_size && col < col_size)
  {
    out[row_size * col + row] = result;
  }
}
```

Table H.3: Comparison between used CodeGen and GPT4. Last example for generating code for matrix multiplication but with the use of ___*shared*___ memory. Both snippets are again very similar and functionally almost identical, except that the GPT4 better structures the code.