Received 22 January 2024, accepted 26 February 2024, date of publication 1 March 2024, date of current version 11 March 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3372853

RESEARCH ARTICLE

Code Detection for Hardware Acceleration Using Large Language Models

PABLO ANTONIO MARTÍNEZ[1], GREGORIO BERNABÉ[2], AND JOSÉ MANUEL GARCÍA[2], (Member, IEEE)
[1]Huawei Technologies Research and Development, CB4 0WN Cambridge, U.K.
[2]Computer Engineering Department, University of Murcia, 30100 Murcia, Spain

Corresponding author: Gregorio Bernabé (gbernabe@um.es)

This work was supported in part by the Ministerio de Ciencia e Innovación (MCIN)/Agencia Estatal de Investigación (AEI)/10.13039/501100011033 under Grant TED2021-129221B-I00 and Grant PID2022-136315OB-I00; in part by "European Union (EU) NextGenerationEU/Plan de Recuperación, Transformación y Resiliencia (PRTR);" and in part by "European Regional Development Fund (ERDF) A way of making Europe," EU.

ABSTRACT Large language models (LLMs) have been massively applied to many tasks, often surpassing state-of-the-art approaches. While their effectiveness in code generation has been extensively studied (e.g., AlphaCode), their potential for code detection remains unexplored. This work presents the first analysis of code detection using LLMs. Our study examines essential kernels, including matrix multiplication, convolution, fast-fourier transform and LU factorization, implemented in C/C++. We propose both a preliminary, naive prompt and a novel prompting strategy for code detection. Results reveal that conventional prompting achieves great precision but poor accuracy (67.5%, 22.5%, 79.5% and 64% for GEMM, convolution, FFT and LU factorization, respectively) due to a high number of false positives. Our novel prompting strategy substantially reduces false positives, resulting in excellent overall accuracy (91.2%, 98%, 99.7% and 99.7%, respectively). These results pose a considerable challenge to existing state-of-the-art code detection methods.

INDEX TERMS Code detection, compilers, heterogeneous computing, high-performance computing, large language model.

I. INTRODUCTION AND MOTIVATION

In recent years, transformer-based models have superseded state-of-the-art neural network models for all kinds of tasks, like computer vision [1], natural language processing (NLP) [33], speech recognition [35] or translation [21]. One of the most impressives models based on the transformer architecture [40] are large language models (LLMs). A LLM is no more than a language model trained with significantly larger training data than traditional language models. In essence, LLMs are probability distributions over sequences of words. That is to say, language models aim to understand and generate text, just like humans do.

LLMs are provoking impressive impact in diverse deep learning fields, thanks to their impressive abilities to perform diverse tasks [17], [42]. The extensive list of applications where LLMs work well implies that there may still be unexplored possibilities awaiting discovery. In particular, even though LLMs are proven to work for code synthesis [42], their evaluation in code detection remains unexplored. Code detection involves analyzing programs to identify the algorithms they contain, which is crucial in numerous areas of programming language research. A notable application of code detection is to match and replace handwritten code with optimized libraries.

Specialized hardware accelerators provide massive performance and energy efficiency improvements over traditional microprocessors [8]. However, there is a lot of code that is already written for CPUs, so executing it on accelerators is not trivial. Detecting parts of acceleratable code and replacing it with appropriate API calls is a novel approach to overcome these issues. This technique has many advantages, like improving the performance of a hardcoded implementation or offloading compute-heavy tasks to accelerators

The associate editor coordinating the review of this manuscript and approving it for publication was Tomás F. Pena.

automatically. Many works focused on this topic [9], [13], [23], [45], which rely either on constrained-based pattern matching or neural network-based code detection. Those code detection techniques are generally either brittle, unable to match complex code, or inefficient in recognizing code patterns. On the contrary, LLMs are trained with a huge corpus of source code, so code detection is a field where LLMs can potentially outstand.

Despite LLM powers, it is still challenging to exploit their full capabilities because they are susceptible to the prompt. LLMs can suffer from different kinds of faults, like hallucination [15], which has motivated the emergence of a new discipline called prompt engineering. Prompt engineering aims to build prompts for maximizing the performance of LLMs. However, depending on the specific application, different prompt engineering methods shall be used because not all work well for every situation.

This paper proposes a novel methodology that leverages LLMs to perform code detection. More specifically, we investigate the use of the GPT-3.5 platform and OpenAI's API for code detection tasks. To evaluate its performance, we build a benchmark suite consisting of significant and computationally intensive algorithms from computer science. Although this evaluation does not cover all possible algorithms, it provides proof of how LLMs perform at code detection. Our evaluation demonstrates that GPT-3.5 successfully detects a 93% of matrix multiplication (GEMM), 100% convolution, 100% fast-Fourier transform (FFT), and 93.3% LU factorization programs. Furthermore, we evaluate GPT-3.5 using a benchmark containing diverse program implementations that do not include the target algorithms. This evaluation reveals a false positive rate of 44%, 80%, 21%, and 37% for GEMM, convolution, FFT, and LU factorization, respectively. We investigate the causes of this issue and propose a novel prompting strategy for code detection using prompt engineering. Using our novel proposal, we significantly reduce the false positive rates to 5.9% for GEMM, 1.6% for convolution, and 0.0% for FFT and LU factorization, significantly improving the accuracy. Our results affirm the viability of LLMs, specifically GPT-3.5, for code detection tasks, no matter what kind of algorithm we want to detect. This indicates that LLMs have a great potential for integration with existing compilation techniques to replace handwritten code with accelerated implementations.

This paper makes the following contributions:

- We present the first analysis on code detection using large language models.
- We propose a novel prompting strategy for code detection that drastically improves LLM performance at code detection tasks.
- We evaluate GPT-3.5 in code detection against matrix multiplication, convolution, and fast-fourier-transform programs, as well as several other programs to evaluate GPT-3.5 false positive rates.

The rest of this paper is organized as follows. Section II presents the background on prompt engineering and large language models, code detection techniques, and their applications for hardware acceleration. We present our methodology to use GPT-3.5 API for code detection in Section III. Section IV shows our prompt engineering work, divided into two approaches, the second being a novel prompting technique for code detection. In Section V we evaluate the accuracy of each prompting technique, showing that GPT-3.5 is indeed a code detection tool for hardware acceleration. Finally, Section VI concludes the work and gives hints and suggestions for future work.

II. BACKGROUND AND RELATED WORK

A. LARGE LANGUAGE MODELS (LLMS)

Large Language Models (LLMs) refer to a family of language models, based on the transformer architecture [40] containing billions of parameters that are trained on massive datasets containing text [42]. Generative Pre-trained Transformer 3 (GPT-3) is a 175 billion parameter LLM released in 2020 by OpenAI [4]. Two years later, OpenAI released a new subclass, GPT-3.5, which is the base of the popular ChatGPT, a chatbot fine-tuned for conversations that have caused a massive shock in society [5], [11]. OpenAI's last release is GPT-4 [32], a new version in the GPT series, which has shown impressive results in many tasks, leading us to think that we are approaching artificial general intelligence (AGI) [5]. Other notable LLMs include Pathways Language Model (PaLM) [6] (Google) or LLaMA [39] (Meta).

Applications where LLM exceeds are uncountable. From coding to mathematical abilities, LLMs' power focuses on natural language tasks. One particular aspect of LLM is their capability to show abilities that are not present in small models but arise in large models [48]. Generally speaking, we can organize these abilities into three domains: in-context learning (ICL), instruction following and step-by-step reasoning (e.g., chain-of-thought) [42], [48]. ICL consists of prompting LLMs with a question in natural language description along with several demonstrations and a test query [42]. In instruction following, tasks are described in the form of instructions. With instruction tuning, LLMs can follow the task instructions for new tasks without using explicit examples, thus improving their ability to generalize. Lastly, step-by-step prompting includes several intermediate reasoning steps, which LLMs can use to perform complex reasoning.

B. PROMPT ENGINEERING

Language models are designed to imitate and predict the next token given a sequence of tokens as input [2], which explains why LLMs are extremely sensitive to the prompt. Thus, it is crucial to understand how to provide good prompts to LLMs in order to achieve good-quality outputs. In this sense, prompt engineering covers a set of techniques to improve the communication between humans and LLMs. Prompts are instructions given to an LLM to specify the quality and quantity of the output, enforce rules in the output, etcetera. Prompting can actually also be considered as a form of programming an LLM since we are giving precise instructions of how and what to do [44]. In that respect, recent works have proposed new languages to aid programming of LLMs [3], [27].

Zero-shot prompting is the most straightforward prompting technique, where the prompt contains only instructions describing the task. On the contrary, few-shot prompting consists of providing additional examples or demonstrations of the model. This technique can improve LLM performance on complex tasks, where zero-shot prompting is not enough.

Like humans, LLMs are thought to be capable of performing reasoning [26]. In this line, one critical prompt engineering technique to improve reasoning is called chain-of-thought (CoT) [43]. With CoT, the prompt is spread out across a larger sequence of tokens. Instead of prompting the model with just the question, CoT prompts include examples of chain-of-thought sequences of the task. In this sense, CoT is an instance of few-shot prompting, proposing a simple solution by modifying the answers in few-shot examples to step-by-step answers [18].

Another relevant technique is called self-consistency [41]. Intuitively, it is based on the idea that we, as humans, think in different ways. In tasks requiring reasoning, it is natural to have several ways to attack the problem. Self-consistency consists in prompting the model using CoT multiple times to sample a diverse set of reasoning paths. Afterward, it selects the most consistent answer by marginalizing all the available answers in the answer set. This method has proved to be effective in several scenarios [41].

A more complex technique, based on the previous approaches of CoT and self-consistency is tree-of-thoughts (ToT) [46]. This paradigm allows LLMs to explore multiple reasoning paths over thoughts. In this context, a thought is considered a part of the final solution. ToT prompting involves defining four key aspects: how to decompose the intermediate process into thoughts, how to generate potential thoughts from each state, how to evaluate each state and what search algorithm to use. This technique has shown promising results in solving complex tasks such as Game of 24, Creative Writing, and Crosswords [46].

A simpler, yet crucial aspect of prompting is controlling the format of the output. For example, it is useful to use delimiters to demarcate sections of text to be treated differently [30]. Other techniques for controlling the output format are specifying the desired length of the output or the format and order in which each part of the output should be presented.

C. PROGRAMMING HARDWARE ACCELERATORS

Specialized hardware accelerators provide massive performance and energy efficiency improvements over traditional microprocessors [8], necessary to overcome the challenges of increasingly complex and compute-demanding applications. Instead of using one device (CPU) for everything, accelerators are specialized for a given domain. The benefit of microprocessors' generality is their ease of adaption, but at the same time, it is their source of inefficiency. However, accelerators are highly diverse [34], which makes programming particularly hard.

To write new code for accelerators, the typical approach is writing software in the programming language specifically designed for that accelerator. For existing code (e.g., a program written in a general-purpose language) there is a better alternative than rewriting the program using the accelerator-specific language. This alternative involves replacing parts of the code with calls to the accelerator API [9], [13]. By utilizing libraries that target the accelerator API, this approach allows the compilation of old code, effectively using the accelerator without the need for extensive code rewriting.

To replace accelerable parts of code with calls to an optimized library, the compiler must accomplish two distinct tasks. First, the compiler must detect parts of the code that are suitable for replacement. Second, the compiler must find the mappings between the variables in the original program and the variables in the library. In this work, we focus on exploiting LLMs to achieve the first task, code detection, which discovers the accelerable parts of the code that are candidates for replacement.

D. CODE DETECTION

Code detection approaches in the literature can be divided into two categories:

- **Constraint-based matching**: Also known as pattern-based matching, consists in finding constraints and patterns in the code that can be matched into a previously defined set of constraints [29]. Idiom Description Language (IDL) [13] proposed a description language that allows the user to define constraints for detecting particular idioms. Idioms are later translated into a set of constraints over the LLVM IR [19], which are used by the compiler to match the corresponding code. In [12], authors also focus on idiom matching and rewriting, but the idiom specification aims to be easier to understand. Besides, idioms are translated into MLIR [20] dialects rather than LLVM IR. Similarly, a pattern constraint-based approach is used in KernelFaRer [9], where authors focus primarily on detecting matrix-multiplication (GEMM) and SYR2K kernels. Here, constraints are hardcoded inside the compiler, and the user can not modify them. Constraint-based matching is, however, very brittle and generally unable to match complex code structures [23].
- **Neural embeddings**: A neural network is trained to detect the code. This is a more modern approach to code detection [23], [45], which uses neural embeddings

to detect accelerateable regions. In [23] and [45], the neural classifier is based on ProGraML [7], and it is trained using the OJClone dataset [28], containing 105 classes of different algorithms, each implemented in different ways. New algorithms not present in the dataset can be easily identified by adding examples of that particular algorithm to the dataset, which will make the neural classifier learn how to identify them. The downside of current neural embedding proposals is their low accuracy. They perform well to guide the search of a given algorithm but although they can match more complex code structures, they are less reliable compared to constraint-based matching.

Above mentioned techniques aid to find and categorize sections of code, but they do not provide any guarantee that those sections are correctly identified, which can potentially lead to faulty compilation. Therefore, some verification is typically coupled with code detection to prove that the code was identified correctly and works as expected. Besides formal verification techniques, other approaches rely on comparing the output of the code with the output of a valid program (input-output validation) [23], [45]. Despite the efforts to ensure the correctness of code, programmer sign-off is ultimately required.

III. USING GPT-3.5 TO ANALYZE CODE

A. USING THE API

At the time of writing, OpenAI's models (GPT-3.5, GPT-4) and Claude are the most competitive general-purpose LLMs [47].[1] Another LLM to consider is AlphaCode [22], a model specifically trained to generate code. However, it is not publicly available, and it is limited to code generation, not code detection. While Claude and GPT-4 are in a limited beta (they are only accessible to those who have been granted access), GPT-3.5 (the base model for ChatGPT) is publicly available [31]. In this work, we use the `gpt-3.5-turbo-16k` model (the most capable GPT-3.5 model, according to OpenAI [31]) with the OpenAI API. The `16k` suffix simply indicates that this model has a 16k context window, in contrast to the standard GPT-3.5 model which has a context window of 4k tokens. Having a larger context window allows to analyze larger codes that otherwise would not fit in 4k or fewer tokens. We believe that the proposals shown in this work are applicable to any other LLM and that results shall be similar.

We designed a simple wrapper around the OpenAI API written in Python. It is responsible for reading the source code, prompting the `gpt-3.5-turbo-16k` model with the code, and parsing the output. As we will see in Section IV, our prompt includes formatting instructions. However, sometimes GPT-3.5 produces outputs that do not match exactly the specified output format. This motivates the need for a simple parser (detailed in Section IV-C), which allows our Python program to interpret the output (e.g., to understand when a program is correctly classified and when it is not). The API can be configured with parameters such as the role of the temperature and nucleus sampling parameters. The role parameter indicates how the model should behave (acting as in a given role, such as system, user, assistant, or function). Temperature and nucleus sampling (`top_p` parameter) allows for control of the sampling of the model. Both are ways to control the sampling, so it is recommended to alter only one of those, but not both. Intuitively, higher temperature values will make the output more random, while lower values will make it more deterministic.

[1]Ranking available at: https://chat.lmsys.org/?leaderboard

B. THE DATASET

In this work, we focus on detecting C/C++ codes, so all codes in our dataset are implemented in one of those languages. We identify three key kernels in code detection works, as well as one uncommon kernel. GEMM is clearly the one that gets most of the attention [9], [13], [23]. Fast-fourier transform (FFT) and convolution are also studied in the literature [23], [45]. We also include LU factorization codes to explore the possibility of extending code detection works to less common kernels, which could challenge GPT-3.5 detection capabilities. To create the dataset, we explored GitHub C/C++ code performing any of those four kernels. To better understand the quality of GPT-3.5 code detection, we focused on finding different implementations of each algorithm.

1) GEMM

We identified and gathered 7 classes of matrix multiplications:

- *Naive:* Implementations with the traditional 3-loop structure.
- *Unrolled:* Implementation with unrolled loops.
- *Function Calls:* Implementations dividing the compute into different function calls.
- *Tiled:* Tiled implementations.
- *Goto:* Implementations using the Goto algorithm [14].
- *Strassen:* Implementations using the Strassen algorithm [37].
- *Intrinsics:* Implementations using Intel intrinsics.

2) CONVOLUTION

We found 3 different implementations:

- *Winograd:* The Winograd algorithm.
- *Direct:* The direct convolution algorithm.
- *im2col+gemm:* Uses a method called *im2col* to compute the convolution using GEMM (e.g., like the Caffe framework [16]).

3) FFT

We retrieved 3 different implementations:

- *DFT:* Discrete fourier transform implementations.

TABLE 1. Source code dataset description (available at [24]).

	Code	Number of Codes	Total LoC
Real Code	GEMM	128	11.3k
	Convolution	15	2.8k
	FFT	15	1.8k
	LU	15	1.9k
	Total	158	16.0k
False Positives	Parboil	10	1.4k
	Caffe	182	42.3k
	ACOTSP	13	2.8k
	cpufetch	22	5.7k
	Total	227	54.2k

- *Radix-2:* Computes the DFTs of the even-indexed and the odd-indexed inputs separately and then combines both.
- *Recursive:* Recursive implementations.

4) LU FACTORIZATION

We analyzed 4 different implementations:

- *Naive:* Naive implementations.
- *Pivoting:* LU factorization with full or partial pivoting.
- *Tiled:* Tiled implementations.
- *Intrinsics:* Implementations using Intel intrinsics.

5) FALSE POSITIVES

Also, we are interested in measuring the exposure of the model to false positives. Therefore, we also included programs not explicitly containing the four aforementioned algorithms. We differentiate this part of the dataset between mainstream and non-mainstream code. For the mainstream code, we included the Parboil benchmark [38], a set of applications for benchmarking the performance and throughput of processors, and Caffe [16], a deep learning framework. Those programs are somewhat popular and it is easy to find similar implementations of those applications in the wild. Besides, the Parboil benchmark contains a matrix-vector multiplication, which can be helpful to understand the sensibility of the model, since it is a kernel very similar to matrix multiplication. For the non-mainstream code, we included cpufetch [10], a program that gathers CPU architecture information, and an Ant Colony Optimization (ACO) implementation [25]. Furthermore, for those codes that unintentionally contained GEMM, convolution, FFT or LU, we removed them from the dataset to make sure that those codes do not contain such algorithms. A description of the dataset is shown in Table 1.

C. FEEDING GPT-3.5 WITH CODE

To feed the model with code, we simply copy and paste the code into the prompt. In other words, we use crude code straight into the model. It is worth noting that LLMs have an input token limit, meaning that they can only process inputs smaller than their limit. If the code is larger than the token limit, we identify two ways of processing the input. First, decreasing the token count of the code. The idea is to reduce the token count without changing the code semantics (e.g., replacing spaces with tabs). Removing comments, removing dead code, or reducing the length of variables and function names are also viable, but they might hurt the model's ability to reason about code. Second, partition the code into smaller parts. For example, partitioning the original program into n partitions. In our case, however, none of the codes surpassed the 16k token limit, so we can safely feed GPT-3.5 with code without additional modifications.

> I want to know if the code below contains any function performing a **algorithm**. Please ignore functions whose definition is not visible.
>
> Desired format:
> Yes: function name (if there is a function).
> No (if there is no function)
>
> Code:““““
> **the actual code**
> ””””

FIGURE 1. First prompt.

IV. PROMPT ENGINEERING

One of the most common tips for good prompting is to start with a simple prompt and then iterate over more complex and complete prompts. Here we describe our prompt engineering process in which we started with a very first prompt and a second version of the prompt that is aimed to improve it.

A. FIRST PROMPT

The first prompt is detailed in Figure 1. In this prompt, the keyword **algorithm** is replaced by the specific algorithm we are looking for. That is, **algorithm** may take the value of ‘‘matrix multiplication (GEMM)’’, ‘‘convolution’’, ‘‘fast Fourier transform (FFT)’’ or ‘‘LU factorization (LU)’’. Also, the keyword **the actual code** is replaced by the code itself.

First, we naturally ask the model to search for the specific algorithm we are interested in. We also ask the model to ignore functions whose code is not visible. In this sense, we are concerned about hallucinations. Because the model has been trained with large code bases, it could have been trained with the same code (at least, parts of the code) that we will be analyzing. This can lead the model to think that it knows the code for unseen functions, which is actually wrong because supposing that a function with the same name and arguments corresponds to another function is no more than an assumption.

Next, we specify the output format. We use ‘‘Desired output’’ to clearly indicate that follows the output format specification. To make the output easy to parse, we ask the model to output ‘‘Yes’’ or ‘‘No’’, followed by the name of the function (or functions) in case the answer is affirmative.

Can you explain what the following code does?

Code:“““
the actual code
”””

FIGURE 2. Second prompt (part 1).

Does the code contain any function performing a *algorithm*? Please ignore functions whose definition is not visible.

Desired format:
Yes: function name (if there is a function).
No (if there is no function)

FIGURE 3. Second prompt (part 2).

Afterward, we simply paste the code between three quotation marks to delimit the beginning and end of the source code.

B. SECOND PROMPT

In the first prompt, we have clearly stated the task to perform. Thus, we expect to have good detection results when the code contains any of the algorithms. However, it is not clear how the prompt allows the discard of false positives or the reduction of hallucination effects. Hence, the second iteration of the prompt tries to accomplish this matter.

We considered using several prompt engineering techniques like chain-of-thought [18], [43], self-consistency [41], or tree-of-thoughts [46]. However, none of these techniques apply to code detection. First, code detection does not follow any reasoning to conclude whether a code corresponds to a class of algorithms or not (it is more of a classification-like task). Second, all of these techniques focus on improving the reasoning capabilities of LLMs in complex tasks, not mitigating hallucinations. Previous works have highlighted that LLMs are able to identify when they have produced a wrong answer [36]. This, however, requires several prompts. The first one contains the task to be performed by the LLM, and the second, where the LLM can use self-reflection to identify whether the previous answer is valid or not. This two-step prompting of describing the task in a first prompt and realizing that it was wrong in a second prompt inspired us to propose a novel prompting technique for code detection.

The second prompt is composed of two phases which are shown in Figures 2 and 3. Rather than following a zero-shot approach (like in the first prompt) here we use two prompts. In the first part, we simply ask the model to explain what is the code doing. Leaving the model to freely explain what the code does work very well because it is very easy for the model to understand what the task is. Most of the time, the explanations given by the model at this step are correct (hallucinations are not present, or at least are very rare). Once the model has analyzed the code, we ask, in a second prompt, if the code contains a given algorithm.

Please note the difference between this and the previous prompt. Previously, we asked directly whether the code contained an algorithm. Now, we ask the model to describe the code and then we ask if in that “description” that the model gave is found the algorithm in question. In essence, false positives may arise in the first prompt (e.g., the model wrongly identifying parts of the code) but it is way less probable than asking directly to check for the algorithm. In contrast, false positives may not appear as a consequence of the second prompt because the model is simply reusing the information previously given. Thus, we expect to reduce hallucination effects with this technique, while maintaining similar detection results.

C. OUTPUT FORMATTING

When the model's output matches exactly the expected output, the wrapper does not perform any parsing. Here is a description of the rules that the wrapper applies when the output does not match:

- **If the expected answer is positive, e.g.: “Yes: (function list)”**: The wrapper removes the following substrings from the output: “\nNo”, “\n”, “.”, “()” (where \n is a new line). This makes it possible to accept outputs that contain outputs containing any “garbage”.
- **If the expected answer is negative, e.g.: “No”**: The wrapper removes the following substrings, which we observed that occasionally appear: “the code does not contain any function”, “there is no function”, “the code does not contain any function” and “there is no function”.

V. EVALUATION

A. SETUP

We evaluate the GPT-3.5 model using the source code dataset shown in Section III-B with the two proposed prompts. More precisely, we use `model='gpt-3.5-turbo-16k'`, `temperature=0.0`, `top_p=1.0` and `max_tokens=512`. The selected model allows inputs up to 16K tokens, allowing us to analyze larger codes, which would not be possible with the default GPT-3.5 model (which supports inputs with up to 4K tokens). We aim to obtain answers as deterministic as possible, so we set `top_p` to 1.0 (the default value) and only modify `temperature`, which we set to 0. Lastly, `max_tokens` sets the maximum number of allowed tokens in the output, which we limit to 512 since it is enough for the first step of the second prompt.

In programs with multiple valid functions (e.g., multiple functions performing a GEMM), we expect the model to find the outermost function. For each prompt, we show the confusion matrix and also a summary matrix that presents

TABLE 2. Confusion matrices for GEMM, CONV, FFT and LU (first prompt).

GEMM

Actual \ Predicted	T	N
T	93% (119)	7% (9)
N	44.5% (121)	55.5% (151)

CONV

Actual \ Predicted	T	N
T	100% (15)	0% (0)
N	80.5% (310)	19.5% (75)

FFT

Actual \ Predicted	T	N
T	100% (15)	0% (0)
N	21.3% (82)	78.7% (303)

LU

Actual \ Predicted	T	N
T	93.3% (14)	6.7% (1)
N	37.1% (143)	62.9% (242)

how each algorithm is classified. In the false positives evaluation for a given algorithm, we also include real programs from other algorithms (e.g., for GEMM, we add convolution, FFT, and LU codes to the false positives dataset). Besides, we provide a detailed explanation of why the model was unable to find the algorithm. We identify three types of errors:

- Error 1: GPT-3.5 thinks there is no function, where there is actually at least one.
- Error 2: GPT-3.5 finds at least one function, but not the one we are looking for (the outermost).
- Error 3: Wrong output format (the output is right, but the Python wrapper is not able to parse it).

B. FIRST PROMPT

Table 2 shows the confusion matrix for the four analyzed algorithms. True positive results are excellent in all cases, achieving 93% in GEMM, 93.3% in LU, and 100% in convolution and FFT codes. This seems to indicate that the model can confidently analyze the code and find if the algorithm is present or not. Conversely, the number of false positives is exceedingly high in the four algorithms, and it is even more notable in the case of GEMM and convolution. Alternatively, we can compute the accuracy of the model as:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

which, using data from Table 2, yields a poor accuracy for Convolution (22.5%), followed by LU (64%), GEMM (67.5%) and FFT (79.5%). Despite the high precision, the accuracy is severely harmed due to the high false positive rate. Results indicate that the model tends to answer our questions affirmatively rather than reasoning about the code and answering accordingly.

TABLE 3. Summary of false negatives types (first prompt).

	GEMM	CONV	FFT	LU
Error 1	2	0	0	0
Error 2	2	0	0	1
Error 3	5	0	0	0
Total	9	0	0	1

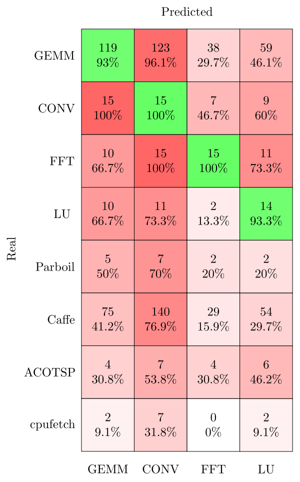


FIGURE 4. Summary of the confusion matrix (first prompt).

Regarding output consistency according to the rules set in the prompt, we found that the rules implemented in the wrapper (described in Section IV-C) are rarely triggered. For the case where the expected output is affirmative, the most simple rules, like removing a dot, are triggered, but not very often. In fact, when the output is negative, rules never get triggered. This indicates that the output formatting rules in the prompt work consistently well since very little parsing is needed.

Table 3 presents the type of false negatives for each algorithm. The most common case is caused by the model not giving a valid output format. Here we mostly found issues with C++ formatting, where our Python wrapper was expecting to find the name of the function, but the model also includes C++ artifacts. The model answered a valid function, but not the outermost function in two cases, while the rest correspond to the model simply answering that there were no functions matching the algorithm.

Regarding false positives, results show that the model gets confused and identifies algorithms where they are not. It is worth noticing that these false positives always occur on functions that are defined and declared in the code. The only exception to this rule is found in Caffe, where we found that the model also triggers false positives in functions that contain "GEMM" in the function name, even if they are not defined or declared. It seems reasonable to think that such functions perform matrix multiplication, but if code is not available it is only an assumption that the model is unable to confirm. This appears to be a clear example of external hallucination since the model thinks to know

TABLE 4. Confusion matrices for GEMM, CONV, FFT and LU (second prompt).

GEMM

Actual \ Predicted	T	N
T	85.2% (109)	14.8% (19)
N	5.9% (16)	94.1% (256)

CONV

Actual \ Predicted	T	N
T	86.7% (13)	13.3% (2)
N	1.6% (6)	98.4% (379)

FFT

Actual \ Predicted	T	N
T	93.3% (14)	6.7% (1)
N	0% (0)	100% (385)

LU

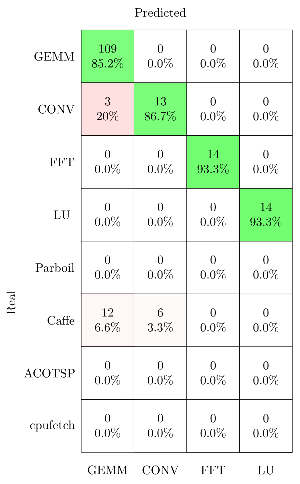
Actual \ Predicted	T	N
T	93.3% (14)	6.7% (1)
N	0% (0)	100% (385)

the content of the function, which is actually unavailable. We also observed clear cases of hallucination in some programs where there was no function to detect but the model gave an output with a length equal to `max_tokens` (e.g., exhausting the output) repeating one function name over and over.

Figure 4 depicts the results summary for the first prompt, showing for each algorithm (GEMM, CONV, FFT, LU) in the x-axis, the percentage of matched programs against each code type (in the y-axis). Ideally, we would like to have as high values as possible for the diagonal of the first four elements while the rest of the cells are as close to zero as possible. Having high values in the diagonal indicates high true positive rates, while low values in the rest of the figure indicate low false positives. In the first prompt, we find high values in the diagonal, as well as in the rest of the figure. The highest false positive rates are found when we ask the model if a GEMM program contains any convolution (96.1%), and when we ask if convolution or FFT programs contain GEMMs (100% in both cases). Results evidence that the model gets easily confused when mixing GEMM and convolution (asking for GEMM in convolution programs and vice versa). This might be motivated by the fact that both have relatively similar code structures and that sometimes convolutions are implemented with matrix multiplications. However, we also find surprisingly high false positives when analyzing other codes. In the Parboil benchmark, we obtain rates as high as 50% and 70% for GEMM and convolution, respectively. This benchmark suite contains programs for evaluating the performance of microprocessors, which also have certain similarities with GEMM and convolution. In Caffe, the false positive rate is also high. The model gets easily confused with this code because many of them have functions containing "gemm" or "conv" in the code. However, most of the time, they are simply function calls rather than function definitions.

TABLE 5. Summary of false negatives types (second prompt).

	GEMM	CONV	FFT	LU
Error 1	1	1	0	0
Error 2	11	1	1	1
Error 3	7	0	0	0
Total	19	2	1	1



Real \ Predicted	GEMM	CONV	FFT	LU
GEMM	109 85.2%	0 0.0%	0 0.0%	0 0.0%
CONV	3 20%	13 86.7%	0 0.0%	0 0.0%
FFT	0 0.0%	0 0.0%	14 93.3%	0 0.0%
LU	0 0.0%	0 0.0%	0 0.0%	14 93.3%
Parboil	0 0.0%	0 0.0%	0 0.0%	0 0.0%
Caffe	12 6.6%	6 3.3%	0 0.0%	0 0.0%
ACOTSP	0 0.0%	0 0.0%	0 0.0%	0 0.0%
cpufetch	0 0.0%	0 0.0%	0 0.0%	0 0.0%

FIGURE 5. Summary of the confusion matrix (second prompt).

It is surprising to find that the model gets confused with these even though we explicitly asked to ignore functions whose definition is not visible. Also, sometimes functions are visible, but they do not perform GEMM or convolutions explicitly in the code. Lastly, we also found surprisingly high false positive values in ACOTSP-MF and cpufetch, even though those programs have in no way any similar code to GEMM or convolution.

C. SECOND PROMPT

Table 4 shows the confusion matrix using the second prompt. The first thing that draws attention is the true positives. They are very similar to the one we had in the first prompt, but they are slightly lower. However, the number of false positives has decreased significantly, with only 5.9%, 1.6%, 0.0% and 0.0% for GEMM, convolution, FFT and LU, respectively. These results confirm that our prompt proposal drastically improves the accuracy of the model, which raises to 91.2%, 98%, 99.7% and 99.7%, respectively.

In the second prompt, we found that the rules implemented in the wrapper (shown in Section IV-C) are much more likely to be triggered than in the first prompt. Specifically, rules for the case where the expected output is negative were never triggered in the first prompt, but they are sometimes triggered in this prompt. These results indicate that the output specification is less robust in the second prompt compared to the first one which showed to be pretty reliable.

We analyze the failure reasons for the second prompt in Table 5. The majority of failures are found in Error 2, (e.g., when the model finds at least one function performing a matrix multiplication, but it is not the one we were expecting to find). Besides, we still have the same problems in GEMM programs where the model does not comply with the output format that we expect, which makes the wrapper unable to parse the output.

In the second prompt, false positives only arise when looking for GEMM and convolution in Caffe, and when looking for GEMM in convolution codes. As we mentioned in the first prompt, the model had two types of false positives in Caffe. The majority were functions declared and defined in the code, while a minority were undefined functions containing "gemm" in the name. In the second prompt, all the false positives from Caffe correspond to the second class. This indicates that the model still hallucinates with functions that have "gemm" in its name, while the other false negatives have completely disappeared. Lastly, in convolution codes, we can still find some cases where the code is purely performing a convolution but the model reports it as performing a matrix multiplication. Sometimes, it happens because the code contains a matrix multiplication inside the reported function, but it does not mean that the function performs matrix multiplication exclusively (which is what we are asking). However, we believe that these issues should be fixed with more powerful models (e.g., GPT-4) or code-specific models (e.g., AlphaCode).

VI. CONCLUSION AND FUTURE WORK

Large Language Model's scale and complexity have grown massively in the last few years. ChatGPT, in particular, and LLMs, in general, have caused a massive impact on society due to the enormous potential of these models for greatly diverse natural language tasks. We also expect those models to keep scaling and improving in the near future, so finding new applications where these models excel is key for fully exploiting them. A field not previously explored with LLMs was code detection, which is key for many applications in programming language research. Particularly, it has been studied to achieve code lifting, a technique that consists in replacing handwritten code with a call to an optimized library. Previous work has approached this topic either with constraint-based matching or with neural embeddings plus input/output equivalence, which can be costly for large codes.

In this work, we have explored the application of LLMs to code detection for the first time. Specifically, we evaluated GPT-3.5 in code detection with matrix multiplication (GEMM), convolution, fast-fourier transform (FFT), and LU factorization algorithms. After designing our first prompt for code detection, the model showed an accuracy of 67.5%, 22.5%, 79.5%, and 64% for GEMM, convolution, FFT, and LU factorization, respectively. False positives, triggered by hallucinations, are the reason to explain such poor results. In the second prompt, we introduced a novel approach for code detection that achieved an accuracy of 91.2%, 98%, 99.7%, and 99.7%, respectively. The new prompt drastically reduces the number of false positives which, still occurring, are way less frequent. Despite not being trained specifically for code detection, GPT-3.5 results are truly impressive, reaching an accuracy very close to 100%.

Rather than using raw code input, we aim to explore alternative approaches in future research. Instead of analyzing program files individually, we are interested in developing a novel methodology that focuses on analyzing the program function by function. This approach would involve creating a dataflow graph that captures the interconnections between the functions defined in the program. Adopting this approach would allow us to analyze the entire code structure as opposed to processing one file at a time. If LLMs are to be embedded with a compiler, then we would need to investigate both inference time and accuracy with local inference models, like Vicuna. Additionally, we are intrigued by the effectiveness of code metadata (function names, variable names and comments) on the result. It may be valuable to understand if the language model would still provide good results if metadata was discarded or if, on the contrary, the language model can achieve similar results without this information. We are also interested in the performance of other models like GPT-4 or other LLMs trained specifically on source code, which may achieve even better results, stretching even more the limit to reach 100% of accuracy.

AVAILABILITY OF DATA AND MATERIALS

The data can be provided upon request.

ACKNOWLEDGMENT

Pablo Antonio Martínez was with the University of Murcia, Murcia, Spain.

REFERENCES

[1] H. Bao, L. Dong, S. Piao, and F. Wei, "BEiT: BERT pre-training of image transformers," 2021, *arXiv:2106.08254*.

[2] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin, "A neural probabilistic language model," *J. Mach. Learn. Res.*, vol. 3, pp. 1137–1155, Mar. 2003.

[3] L. Beurer-Kellner, M. Fischer, and M. Vechev, "Prompting is programming: A query language for large language models," *Proc. ACM Program. Lang.*, vol. 7, pp. 1946–1969, Jun. 2023, doi: 10.1145/3591300.

[4] B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, and S. Agarwal, "Language models are few-shot learners," 2020, *arXiv:2005.14165*.

[5] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg, H. Nori, H. Palangi, M. T. Ribeiro, and Y. Zhang, "Sparks of artificial general intelligence: Early experiments with GPT-4," 2023, *arXiv:2303.12712*.

[6] A. Chowdhery et al., "PaLM: Scaling language modeling with pathways," 2022, *arXiv:2204.02311*.

[7] C. Cummins, Z. Fisches, T. Ben-Nun, T. Hoefler, M. O'Boyle, and H. Leather, "ProGraML: A graph-based program representation for data flow analysis and compiler optimizations," in *Proc. 38th Int. Conf. Mach. Learn. (ICML)*, Jul. 2021, pp. 2244–2253.

[8] W. J. Dally, Y. Turakhia, and S. Han, "Domain-specific hardware accelerators," *Commun. ACM*, vol. 63, no. 7, pp. 48–57, Jun. 2020, doi: 10.1145/3361682.

[9] J. P. L. De Carvalho, B. Kuzma, I. Korostelev, J. N. Amaral, C. Barton, J. Moreira, and G. Araujo, ''KernelFaRer: Replacing native-code idioms with high-performance library calls,'' *ACM Trans. Archit. Code Optim.*, vol. 18, no. 3, pp. 1–22, Jun. 2021, doi: 10.1145/3459010.

[10] D. Noob, ''CPUfetch: Simple yet fancy CPU architecture fetching tool,'' Tech. Rep., 2023, doi: 10.5281/zenodo.7902464.

[11] Y. K. Dwivedi et al., '''So what if ChatGPT wrote it?' Multidisciplinary perspectives on opportunities, challenges and implications of generative conversational AI for research, practice and policy,'' *Int. J. Inf. Manag.*, vol. 71, Aug. 2023, Art. no. 102642, doi: 10.1016/j.ijinfomgt.2023.102642.

[12] V. Espindola, L. Zago, H. Yviquel, and G. Araujo, ''Source matching and rewriting for MLIR using string-based automata,'' *ACM Trans. Archit. Code Optim.*, vol. 20, no. 2, pp. 1–26, Mar. 2023, doi: 10.1145/3571283.

[13] P. Ginsbach, T. Remmelg, M. Steuwer, B. Bodin, C. Dubach, and M. F. P. O'Boyle, ''Automatic matching of legacy code to heterogeneous APIs: An idiomatic approach,'' *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 139–153, Mar. 2018, doi: 10.1145/3296957.3173182.

[14] K. Goto and R. A. V. D. Geijn, ''Anatomy of high-performance matrix multiplication,'' *ACM Trans. Math. Softw.*, vol. 34, no. 3, pp. 1–25, May 2008, doi: 10.1145/1356052.1356053.

[15] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, and P. Fung, ''Survey of hallucination in natural language generation,'' *ACM Comput. Surv.*, vol. 55, no. 12, pp. 1–38, Mar. 2023, doi: 10.1145/3571730.

[16] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, ''Caffe: Convolutional architecture for fast feature embedding,'' 2014, *arXiv:1408.5093*.

[17] J. Kocon, I. Cichecki, O. Kaszyca, M. Kochanek, D. Szydlo, J. Baran, J. Bielaniewicz, M. Gruza, A. Janz, K. Kanclerz, A. Kocon, B. Koptyra, W. Mieleszczenko-Kowszewicz, P. Milkowski, M. Oleksy, M. Piasecki, L. Radlinski, K. Wojtasik, S. Wozniak, and P. Kazienko, ''ChatGPT: Jack of all trades, master of none,'' 2023, *arXiv:2302.10724*.

[18] T. Kojima, S. Shane Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, ''Large language models are zero-shot reasoners,'' 2022, *arXiv:2205.11916*.

[19] C. Lattner and V. Adve, ''LLVM: A compilation framework for lifelong program analysis & transformation,'' in *Proc. Int. Symp. Code Gener. Optim.*, 2004, pp. 75–86, doi: 10.1109/cgo.2004.1281665.

[20] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, ''MLIR: Scaling compiler infrastructure for domain specific computation,'' in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, Feb. 2021, pp. 2–14, doi: 10.1109/CGO51591.2021.9370308.

[21] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, ''BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension,'' in *Proc. 58th Annu. Meeting Assoc. Comput. Linguistics*, Jul. 2020, pp. 7871–7880, doi: 10.18653/v1/2020.acl-main.703.

[22] Y. Li et al., ''Competition-level code generation with AlphaCode,'' *Science*, vol. 378, no. 6624, pp. 1092–1097, Dec. 2022, doi: 10.1126/science.abq1158.

[23] P. A. Martínez, J. Woodruff, J. Armengol-Estapé, G. Bernabé, J. M. García, and M. F. P. O'Boyle, ''Matching linear algebra and tensor code to specialized hardware accelerators,'' in *Proc. 32nd ACM SIGPLAN Int. Conf. Compiler Construction*, New York, NY, USA, Feb. 2023, pp. 85–97, doi: 10.1145/3578360.3580262.

[24] P. A. Martínez, G. Bernabé, and J. M. García, ''GPT codes dataset,'' Tech. Rep., 2023, doi: 10.5281/zenodo.8154364.

[25] P. A. Martínez and J. M. García, ''ACOTSP-MF: A memory-friendly and highly scalable ACOTSP approach,'' *Eng. Appl. Artif. Intell.*, vol. 99, Mar. 2021, Art. no. 104131, doi: 10.1016/j.engappai.2020.104131.

[26] G. Mialon, R. Dessì, M. Lomeli, C. Nalmpantis, R. Pasunuru, R. Raileanu, B. Rozière, T. Schick, J. Dwivedi-Yu, A. Celikyilmaz, E. Grave, Y. LeCun, and T. Scialom, ''Augmented language models: A survey,'' 2023, *arXiv:2302.07842*.

[27] Microsoft. (2023). *Guidance: A Guidance Language for Controlling Large Language Models*. Accessed: Jun. 2023. [Online]. Available: https://github.com/microsoft/guidance

[28] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, ''Convolutional neural networks over tree structures for programming language processing,'' in *Proc. 13th AAAI Conf. Artif. Intell.*, 2016, pp. 1287–1293, doi: 10.1609/aaai.v30i1.10139.

[29] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Berlin, Germany: Springer, 2010, doi: 10.1007/978-3-662-03811-6.

[30] OpenAI. *GPT Best Practices*. Accessed: Jun. 2023. [Online]. Available: https://platform.openai.com/docs/guides/gpt-best-practices

[31] OpenAI. *GPT Models*. Accessed: May 2023. [Online]. Available: https://platform.openai.com/docs/models

[32] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, and R. Avila, ''GPT-4 technical report,'' 2023, *arXiv.2303.08774*.

[33] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, and R. Lowe, ''Training language models to follow instructions with human feedback,'' 2022, *arXiv:2203.02155*.

[34] B. Peccerillo, M. Mannino, A. Mondelli, and S. Bartolini, ''A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives,'' *J. Syst. Archit.*, vol. 129, Aug. 2022, Art. no. 102561, doi: 10.1016/j.sysarc.2022.102561.

[35] A. Radford, J. Wook Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever, ''Robust speech recognition via large-scale weak supervision,'' 2022, *arXiv:2212.04356*.

[36] N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. Narasimhan, and S. Yao, ''Reflexion: Language agents with verbal reinforcement learning,'' 2023, *arXiv:2303.11366*.

[37] V. Strassen, ''Gaussian elimination is not optimal,'' *Numerische Math.*, vol. 13, no. 4, pp. 354–356, Aug. 1969, doi: 10.1007/bf02165411.

[38] J. A. Stratton, C. Rodrigues, I. J. Sung, N. Obeid, L. W. Chang, N. Anssari, G. D. Liu, and W. W. Hwu, ''Parboil: A revised benchmark suite for scientific and commercial throughput computing,'' *Center Reliable High-Perform. Comput.*, vol. 127, p. 27, Mar. 2012.

[39] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, ''LLaMA: Open and efficient foundation language models,'' 2023, *arXiv:2302.13971*.

[40] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, ''Attention is all you need,'' in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30. Curran Associates, 2017, pp. 1–11.

[41] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou, ''Self-consistency improves chain of thought reasoning in language models,'' 2022, *arXiv:2203.11171*.

[42] W. X. Zhao et al., ''A survey of large language models,'' 2023, *arXiv.2303.18223*.

[43] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, ''Chain-of-Thought prompting elicits reasoning in large language models,'' 2022, *arXiv:2201.11903*.

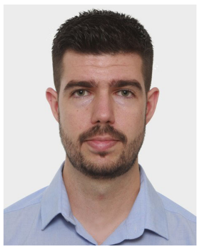
[44] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, ''A prompt pattern catalog to enhance prompt engineering with ChatGPT,'' 2023, *arXiv:2302.11382*.

[45] J. Woodruff, J. Armengol-Estapé, S. Ainsworth, and M. F. P. O'Boyle, ''Bind the gap: Compiling real software to hardware FFT accelerators,'' in *Proc. 43rd ACM SIGPLAN Int. Conf. Program. Lang. Design Implement.*, New York, NY, USA, Jun. 2022, pp. 687–702, doi: 10.1145/3519939.3523439.

[46] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan, ''Tree of thoughts: Deliberate problem solving with large language models,'' 2023, *arXiv:2305.10601*.

[47] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. P. Xing, H. Zhang, J. E. Gonzalez, and I. Stoica, ''Judging LLM-as-a-judge with MT-bench and chatbot arena,'' 2023, *arXiv:2306.05685*.

[48] B. Zoph, C. Raffel, D. Schuurmans, D. Yogatama, D. Zhou, D. Metzler, E. H. Chi, J. Wei, J. Dean, L. B. Fedus, M. P. Bosma, O. Vinyals, P. Liang, S. Borgeaud, T. B. Hashimoto, and Y. Tay, ''Emergent abilities of large language models,'' *Trans. Mach. Learn. Res. (TMLR)*, pp. 1–30, Aug. 2022.



PABLO ANTONIO MARTÍNEZ received the M.S. and Ph.D. degrees in computer science from the University of Murcia, in 2020 and 2023, respectively. During the Ph.D. degree, he was an Assistant Professor with the University of Murcia, where he taught computer architecture for two years. He is currently with Huawei Technologies Research and Development, U.K., as a Compiler Engineer. He has published in several high-impact refereed journals and conferences in these fields, such as *The Journal of Supercomputing*, *Engineering Applications of Artificial Intelligence*, and the International Conference on Compiler Construction. His research interests include high-performance computing, acceleration of deep learning applications, compilers, and heterogeneous computing.

GREGORIO BERNABÉ received the M.S. and Ph.D. degrees in computer science from the University of Murcia, Spain, in 1997 and 2004, respectively. In 1998, he joined the Computer Engineering Department, University of Murcia, and became an Associate Professor, in May 2004. He is currently working on two main research lines: Heterogeneous computing using CMP architectures, GPUs, accelerators (XPUs), and deep neural network learning processes using HPC techniques to improve the performance of medical applications. He has developed several courses on computer structure and computer architecture. He has published more than 50 refereed papers in different journals and conferences, including JCR journals, such as *The Journal of Supercomputing*, *Computer Methods and Programs in Biomedicine*, *Scientific Programming*, *International Journal of Parallel Programming*, and *Journal of Computational Science*.

JOSÉ MANUEL GARCÍA (Member, IEEE) is currently a Professor of computer architecture with the University of Murcia, Spain, and a Founding Member and the Head of the Research Group on Parallel Architecture and Computing (GACOP). His research interest has always focused on the topic of supercomputing. It started within the line of the interconnection network for high-performance computing systems. Subsequently, research work was carried out on issues of improving the performance of single-chip multiprocessors (CMPs), paying particular attention to the internal architecture of the chip, the coherence of the caches, and virtualization. He is working on two main research lines: Heterogeneous computing using CMP architectures, GPUs, accelerators (XPUs), acceleration of bio-inspired algorithms, bioinformatics algorithms, and deep neural network learning processes using HPC techniques. He has directed nineteen Ph.D. theses and published more than 150 refereed papers in international journals and conferences. He is a member of HiPEAC and European Network of Excellence in Architecture and High-Performance Integrated Compilation. Furthermore, he is also a member of various international associations, such as ACM.