

LLM4VV: Developing LLM-Driven Testsuite for Compiler Validation

Christian Munley, Aaron Jarmusch, Sunita Chandrasekaran

Department of Computer & Information Sciences, University of Delaware, 19716, Newark, U.S.A.

Abstract

Large language models (LLMs) are a new and powerful tool for a wide span of applications involving natural language and demonstrate impressive code generation abilities. In this paper, we explore the capability of state-of-the-art LLMs, including closed-source options like OpenAI's GPT-4 and open-source alternatives like Meta AI's Codellama, to automatically generate tests and use these tests to validate and verify compiler implementations of a directive-based programming paradigm, OpenACC. Our approach entails exploring various prompt engineering techniques including a code template, retrieval-augmented generation (RAG) with code template, expressive prompt using RAG with code template, one-shot example, and RAG with one-shot example. This paper focusses on (a) exploring the capabilities of the latest LLMs for code generation, (b) investigating prompt and fine tuning methods, and (c) analyzing the outcome of LLMs generated tests.

Keywords: *Large Language Models, Code Generation, Validation and Verification, OpenACC*

1. Introduction

Large language models (LLMs) are capable of understanding natural language as input and performing tasks with that understanding. LLMs are capable of an array of tasks, from text generation to sentiment classification, code generation, document summarization, and more. OpenAI's GPT-4 scored 90th percentile on the Uniform Bar Exam, 99th percentile on the verbal GRE [1], among several other impressive scores on prestigious academic exams. Similarly, Anthropic's Claude 2 achieved a 76.5% score on the multiple choice section of the Bar exam, and over 90th percentile on GRE reading and writing exams [2]. These models are pre-trained on large, unlabeled datasets with self-supervised learning and can be fine-tuned with domain-specific datasets for improved performance on specific tasks. Code-specific LLMs are trained on a large corpus of open-source code, such as stack overflow and GitHub, enabling the model to write functional and complex code, such as shown in Github Copilot and Codeium [3, 4].

With the vast capabilities of LLMs, the work presented in this paper explores the applicability of the most suitable LLMs and prompt engineering techniques for an optimal generation of a validation and verification testsuite for high-performance computing (HPC) compilers from a standard specification which is mainly natural language. More specifically, the goal of this approach is to check for correctness of C/C++/Fortran compiler implementations of directive-based programming models, OpenMP [5] and OpenACC [6]. To the best of our knowledge, this paper is the first of its kind. While this paper particularly focuses on the directive-based programming model, OpenACC, the approach can be adapted to the OpenMP model as well (this project is also work in progress). The performance of LLMs in programming tasks such as interview-style questions has been explored over the past couple of years [7]. However, the appli-

cation of the approach to the validation of compilers based on a standard specification has not been explored yet.

Briefly, OpenACC, is a directive-based programming model, which targets x86 architectures, A64FX and accelerators such as GPUs. A specification defines OpenACC [6]. Open-source compilers such as GCC [8], LLVM/Clang [9], and vendors such as HPE Cray (for Fortran), and NVIDIA interpret the specification to develop suitable compilers. GCC, Clang, and HPE Cray compilers target X86, NVIDIA and AMD accelerator architectures, while NVIDIA compilers target X86, NVIDIA GPUs and A64FX architectures. OpenACC offers varying levels of control over the execution of a program and the flow of data to users. The model allows the compiler to make optimization decisions or allows for customized distribution of data and execution. This paradigm is beneficial for domain scientists who do not want to re-implement potentially complex domain applications in a low-level parallel programming language such as CUDA [10], OpenCL [11], SYCL [12]. OpenACC is a widely popular programming model spanning several domains including climate [13, 14, 15], computational fluid dynamics [16], nuclear physics [17], biophysics [18] among several other domains.

1.1. Motivation

With the popularity of the programming model, it is critical that the compiler implementations compile and execute the code correctly. Why is this a challenge? Different compiler developers tend to interpret the specification differently often leading to misinterpretations and ambiguities. The complexity of the definition of the features is one of the several reasons. Therefore the compiler implementations are prone to issues or in other words inaccuracy. The other obvious issues include compilation failures or execution failures that need to be identified and addressed. Also each compiler implementation offers

varying levels of support for OpenACC and the coverage of the specification by a compiler varies too, for example HPE/Cray supports only Fortran. Nevertheless, the user should not be burdened with the above challenges nor should they be expected to tailor the usage of the implementations for their applications. So, the expectation is that the implementations should be the same at the level of abstraction for the user.

These above challenges directly call for the need to validate the compiler implementations. This can be done so by writing functional, corner, regression and stress tests. This can be an involved and recurrent task. We have been developing a validation and verification (V&V) testsuite [19] for OpenACC just like we have been developing a testsuite for OpenMP [20].

The task of designing those tests can be quite challenging and cumbersome, and still may not always be correct or complete. There are many ways that the features of the base language, such as C, C++, or Fortran, and the parallel programming paradigm, such as OpenACC or OpenMP, can be used together, and it is humanely impossible to cover them all. Tests in both the OpenACC and OpenMP testsuites are manually created, requiring the teams to stay up-to-date with the tests' generation and maintenance. The team members evolve during the project period. There is a lag when team members leave the project and new members are trained to create quality tests. The lag can range from three weeks to three months or more at times. This can harm the project as we lose continuity and are behind on deliverable. The above hardships serve as a motivation for this work that explores the possibility of using LLMs for a V&V testsuite that can help with complementing the efforts to write these tests manually.

Developers can "shift" focus to designing complicated corner case tests and stress test compiler implementations. LLMs require oversight but can be handy if the effort is front-loaded. LLMs can also help adapt an existing test to new or updated features and perform well at translation between languages. There are many things that LLMs offer developers, though they must be used with human oversight and caution.

This paper makes the following contributions:

- Provides LLM-generated tests to check for correctness of OpenACC compiler implementations
- Explores a fine-tuning dataset for this task
- Evaluates performance of LLMs, choice of prompt engineering techniques, and task-specific fine-tuning on test generation for OpenACC

This paper's generated testsuites, prompts, fine-tuning dataset, and specification json are available at our GitHub [21].

2. Overview of LLMs

Large language models are based on the transformer architecture introduced in [22]. These models are trained on a large corpus of data, generally gathered from the internet. LLMs are trained in two stages: pre-training, and fine-tuning. Pre-training uses a large and general data set, while fine-tuning requires a

```

1 User: Write a code in C to verify compiler
    implementation of the OpenACC specification
    of the acc delete routine.
2 Assistant:
3 ...
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <openacc.h>
7
8 int main() {
9     const int N = 1000;
10    float *a = (float*)malloc(N*sizeof(float));
11    float *b = (float*)malloc(N*sizeof(float));
12
13    // Initialize the arrays
14    for (int i = 0; i < N; i++) {
15        a[i] = i * 1.0f;
16        b[i] = 0.0f;
17    }
18
19    #pragma acc data copy(a[0:N])
20    {
21        // Update 'a' array on the device
22        #pragma acc parallel loop
23        for (int i = 0; i < N; i++) {
24            a[i] = a[i] + 1.0f;
25        }
26
27        // Delete the 'a' array from the device
28        #pragma acc exit data delete(a[0:N])
29
30        // Attempt to update 'a' array on the
31        device again
32        #pragma acc parallel loop
33        for (int i = 0; i < N; i++) {
34            a[i] = a[i] + 1.0f;
35        }
36    }
37
38    // Check the result on the host
39    for (int i = 0; i < N; i++) {
40        if(a[i] != i + 1.0f) {
41            printf("Validation failed at index %
42            d! Host: %f\n", i, a[i]);
43            free(a);
44            free(b);
45            return EXIT_FAILURE;
46        }
47    }
48    printf("Validation passed!\n");
49
50    free(a);
51    free(b);
52
53    return EXIT_SUCCESS;
54 }

```

Figure 1: Example prompt and generated test, with no augmentation through context retrieval, one-shot example, or a code template. This test is a false pass, because it doesn't test the acc routine directive, but the exit data directive.

task-specific data set. Both stages involve optimizing the LLMs parameters by predicting the next word, or token, in the training data set, and updating the parameters, or the weights and biases of the architecture, according to the difference between the actual and predicted next token.

Powerful LLMs consist of a large number of parameters, on

Model / Benchmark	HumanEval pass@1	MBPP
Codelama-34B-Instruct	41.5	57.0
Phind-Codellama-34b-v2	73.8	-
GPT-3.5 (few-shot)	48.1	52.2
GPT-4 (few-shot)	67.0	-

Table 1: Selected LLMs performance on code generation benchmarks

the order of billions. Closed-source LLMs such as OpenAI’s GPT-4 [1] and Anthropic’s Claude [23], are available through a web service or an API, but the parameters are not published. Open-source LLMs such as Meta AI’s family of Codellama models [24] and Stanford’s Alpaca [25], are publicly available for local use, meaning the parameters are published and available for download. Some organizations choose to provide an infrastructure for using the LLM locally, such as for Codellama [24] while many are available on the Huggingface Hub, a community-driven platform with over 120k models [26].

The input to an LLM is text, which must first be tokenized or split into chunks of around four characters [27]. The input text is referred to as a prompt. Once tokenized, the input consists of a set of tokens, from which the LLM creates embeddings, which represent the meaning of the tokens in a high-dimensional vector space. Next, the input embeddings are augmented with positional encodings, because the ordering is not captured in the data otherwise. The LLM uses the whole set of embeddings in the decoding process to produce output. The basic function of a language model is next token prediction - given a set of input data, a prediction for the next most probable token is made. This process is referred to as causal language modeling, but it is not the only task LLMs are capable of. They also excel in various specialized tasks, such as sentiment classification (e.g., rating how positive or negative a statement is), question answering, summarization, and code generation [28].

2.1. Prompt Engineering Techniques

To achieve high-quality performance on complex tasks, specialized input, or prompts, are often required. The most simple prompt is one simply requesting the desired behavior, e.g. "Write a code in C to validate compiler implementation of OpenACC parallel construct." would be a simple prompt. An example of the output from this type of prompt is shown in Figure 1. The example test here passes, but it does not target the correct feature, the acc delete routine in this case, so it is a false pass.

The amount of detail in the prompt regarding the task itself is an important factor in creating an effective prompt. The LLM does not have any prior information about the task, so unless specifically instructed to act a certain way, there is no reason to expect certain behaviour. In this work, we compare the effectiveness of a simple, one-sentence prompt, versus a prompt using multiple sentences describing in detail the task and the requirements. We call the detailed prompt in this work an "expressive prompt".

LLMs are capable of learning to complete a new task based on a few examples provided in the prompt, known as few shot

prompting [29]. For example, if the task is translation, the prompt could include a few example translations before requesting translation of another sentence. This technique can be limited to a single example, known as one-shot prompting. If the task is code generation for interview-style programming questions, a good solution to an example question would be provided as a one-shot example. Alternatively, providing no example of the desired behavior in the prompt is called zero-shot prompting. Few or one-shot prompting can teach the LLM the desired behavior from the prompt, rather than from specific training. This enables a single, general model to be applied to various specialized tasks without additional training for each task. Even a single example can provide the model with information on how to perform the task properly.

Retrieval-augmented generation (RAG) is a common technique to engineer effective prompts by including relevant context to the task, retrieved with a search algorithm, in the prompt [30]. Typically, this is implemented by splitting a text database into chunks, creating embeddings from the chunks that represent the meaning, and storing the embeddings in a vector database. Then, the vector database can be searched for similar information to the prompt, and the retrieved data can be wrapped into the prompt with a template. For example, in a domain-specific question answering task, Wikipedia pages from the domain can be converted into a vector database, and queried for relevant information to each question, to be included in the prompt as context.

Other strategies exist to improve LLM output through prompt engineering, such as chain-of-thought prompting, in which the LLM is instructed to plan its solution to the task, before completing the task [31]. This method can enable the LLM to tackle more complex tasks by planning them out before performing them. A more advanced approach involves creating a parametrized prompt that can be optimized to produce the desired output through training, known as prompt tuning [32]. Prompt engineering is new and broad area, so the optimal methods are still being learned and there are no gold standards that indicate which prompt would be an idea choice for a given case study.

For our work, we explore expressive prompting, one-shot prompting, and RAG. We choose to evaluate these methods for a few reasons. First, they have not previously been evaluated on this task. Next, all three methods are used in order to improve the quality of test generation in different ways. The goal of using expressive prompts in this research is to evaluate the effect of the length of the task description in the prompt on the quality of output. Creating a validation test for OpenACC features is not a simple task, so an expressive prompt seems necessary. We evaluate one-shot prompting as a method that attempts to teach the LLM the proper style of testing OpenACC features, without additional fine-tuning. Finally, we use RAG in this research because its crucial that the LLM is aware of the latest OpenACC specification to create accurate tests, so it must be included either through training or RAG.

2.2. Fine-tuning of LLMs

Prompt engineering is a powerful method to adapt a model to specific tasks without requiring expensive retraining. While prompts are useful, they do not alter the underlying parameters of the language model, thus constraining their ability to specialize or improve performance. Fine-tuning is training a foundation model on a domain-specific dataset, which involves updating all of the model’s parameters. This method can teach a LLM to solve a new task without examples in the prompt [33]. Fine-tuning can also be more robust than prompt design, as the model can be shown more examples than can fit in the prompt. Additionally, updating the LLMs parameters during fine-tuning adapts the model to a specific task, while relying on a prompt leaves the LLMs architecture unchanged. The maximum input size to a LLM, sometimes called the context window, is limited, while the training datasets can be enormous. LLMs can’t see all of the fine-tuning data during inference, they can only operate based on their parameters and input. Using prompts offer the advantage of making all the data directly available to the language model. Though fine-tuning can be costly, parameter-efficient methods can reduce costs, such as freezing the model weights during fine-tuning and introducing small trainable layers into the architecture [34, 35].

LLMs are trained on a large corpus of data from the internet, but that does not necessarily include a lot of information about the subject of our interest, i.e., OpenACC. In order for the LLMs to correctly use and validate OpenACC implementations, they need to learn the features of OpenACC and their definitions. This can be achieved through context within the prompt or through fine-tuning on an OpenACC-specific dataset.

2.3. LLM Benchmarks

Performance of LLMs is evaluated on a variety of benchmarks, often task-specific, such as shown in Table 1. This table focuses on results from the selected LLMs that we have used for our research. There is currently no ‘gold standard’ benchmark suite code generation task, let alone compiler test generation from natural language specification. We choose to consider HumanEval and MBPP benchmarks in the selection of LLMs [36, 37]. HumanEval is an evaluation set released by OpenAI to evaluate functional correctness of Python code generation from docstrings. The HumanEval benchmark is relevant to this work as it evaluates code generation performance from natural language specification at varying levels of complexity. Although the benchmark’s scope does not directly cover test generation quality and feature coverage, it is relevant to the evaluation of code generation from natural language prompts. MBPP is similar to HumanEval, designed to evaluate the performance of LLMs on python programming tasks suitable for entry-level programmers.

The research in this paper focuses on code generation in C, C++, and Fortran, so the above benchmarks discussed provide a baseline for measuring model performance on the general code generation task from natural language prompts.

3. Related Work

We will start off this section by sharing our own on-going and relevant OpenACC V&V testsuite project [19] where we have been manually generating functional tests. This work stands as a good point of reference for the development of LLM work discussed in the rest of the paper. Quite similar to the OpenACC V&V is our other on-going project on OpenMP offloading V&V [20] which also serves the similar purpose of manually generating functional tests. While this current work focuses on OpenACC V&V, work is in progress to adapt lessons learnt for OpenMP V&V as well.

We next summarize some of the relevant and most recent work in this area of research. Many models have been developed for code generation. Meta released a family of Codellama LLMs, OpenAI released Codex and GPT-4, among others, Github created Copilot [3], Amazon created Codewhisper [38], BigCode created StarCoder [39], Microsoft created WizardCoder [40], to name some. Fine-tuned code generation models from foundation models are also prevalent in the open-source community, such as Phind’s fine-tuned models.

Some related works have tried other approaches to improve LLM performance for high-performance computing tasks. In LM4HPC [41], Chen et. al. create a HPC-specific tokenizer, designed to more accurately split the code into tokens corresponding to the typical syntax of HPC code. Moreover, LM4HPC presents multiple HPC-specific training datasets. These are useful both in constructing large pretraining datasets or for domain-specific fine-tuning. LM4HPC also develops three pipelines, code similarity analysis, parallelism detection, and OpenMP question answering, and creates leaderboards from evaluation of performance on these tasks.

Kadosh et al. [42] present Tokompiler, also a tokenizer designed for HPC tasks. Godoy et al. [43] evaluates LLM performance on code generation for HPC tasks. The authors in [44] explore fine-tuning LLMs with HPC-specific data and testing on downstream tasks such as code completion, OpenMP labeling, and performance prediction.

At Microsoft, LLMs are used to create CodeT [45], a method that involves first generation of test cases for code solutions to programming tasks, then execution of the code samples using the generated tests, and evaluation of output. Tufano et al. [46] at Microsoft present AthenaTest, an approach using the BART transformer to generate unit tests. Shafer et al. [47] present a approach, TestPilot, that enables the LLM to attempt to fix failing tests.

LLMs are being gradually adopted for HPC problems. While the literature review refers to test generations using LLMs, they are not necessarily compiler validation tests, but provide a similar workflow. Our research stands apart from these projects in that it evaluates the performance of LLMs on a) comprehending a lengthy natural language programming paradigm specification, and b) subsequently generating validation tests for compiler implementations. This paper, to the best of our knowledge, is the first in this direction.

4. Methods

This section provides a detailed description of how we used LLMs, prompt engineering techniques and datasets to automatically generate and evaluate tests for OpenACC. Our choices were grounded in recent benchmarks and related work.

```
1 f'''Write a code in {language} to verify
2   compiler implementation of the OpenACC
3   specification of {feature}.
4
5   Make sure to follow the template of the format
6   provided. Include the provided header file,
7   and any other necessary libraries.
8   Write simple code to test {feature} in {language
9   }. Try to isolate that feature while still
10  using it correctly.
11  This code is part of a testsuite that will be
12  ran, so write complete code, don't leave it
13  unfinished.
14  The goal is to return 0 if the target feature, {
15  feature}, is working properly, and not zero
16  otherwise.
17  The context below is from the most recent
18  OpenACC specification, make sure to refer to
19  it to produce up to date code.
20
21  Context: {context}
22
23  Template: {template}
24  '''
```

Figure 2: Expressive prompt with RAG and Template. This prompt includes detailed instructions on the task, and provides context from the OpenACC specification and a code template below.

4.1. Selection and initial expectations from LLMs

In this subsection, we somewhat set the stage on what to expect out of the LLMs that we would use for our work so that we can eventually decipher how close or far we were from our expectations. Based on the benchmarks described in Section 2.3 and performance tabularized in Table 1, we selected the following LLMs for this research: OpenAI’s GPT-3.5-Turbo and GPT-4, Meta AI’s Codellama-34b-Instruct, and Phind’s fine-tuned Codellama-34b-Instruct v2 [29, 1, 24, 48]. There are no verified sources for OpenAI that we found that provides information on the size of GPT-4, in other words, the number of parameters. The largest version of GPT-3 is 175B parameters, much larger than Codellama-34B. As GPT-4 is GPT-3’s successor, GPT-4 is presumably larger than any other model and scores high on a variety of benchmarks, so we initially expect the best performance from this model. However, Phind-Codellama-34b-v2 scores highest on the HumanEval benchmark as shown in Table 1. Additionally, Codellama models were trained specifically for code generation, while GPT-4 is general purpose, so we expect Codellama-34b-Instruct and Phind-Codellama-34b-v2 to be competitive.

4.2. Prompt creation

To create V&V tests for validation of OpenACC compilers using LLMs, we create sets of prompts requesting a test for each

feature of OpenACC as listed in Chapters 2 and 3 of OpenACC specification’s table of contents [6]. Please refer to Section 2.1 for an overview of prompt engineering techniques. To explore the effectiveness of an expressive prompt, retrieval-augmented generation (RAG) and one-shot prompting methods, we create multiple sets of prompts both incorporating and excluding these methods. For all tests without one-shot examples, we choose to provide a code template in the prompt, to instruct the LLMs to produce standardized output for testing and evaluation. Thus, we create the 5 following sets of prompt methods and in the upcoming subsections, we narrate them in detail:

- code template
- RAG with code template
- expressive prompt using RAG with code template
- one-shot example
- RAG with one-shot example

4.3. Retrieval-Augmented Generation (RAG)

To provide relevant context from the OpenACC specification in each prompt, we implement RAG using two methods: a similarity search algorithm, and manual retrieval of context with a JSON of the specification. Providing the most recent OpenACC specification to the LLMs is essential to testsuite generation because we do not know that the LLMs are pre-trained on the latest version of the specification, if any. Moreover, if pre-trained on the latest specification, the LLMs can only access information from the specification through the parameters of the model which are optimized during training. The training data is not perfectly stored in the parameters, leading to LLMs producing false facts, known as hallucinations [49].

To enable accurate factual recall, RAG methods are implemented to provide the LLMs with direct access to relevant facts during inference. In this work, we provide relevant pieces of the specification in the prompt to the LLMs to evaluate the effectiveness of RAG for improving LLMs ability to recall the specification accurately, rather than improperly use OpenACC in the generated test.

To manually retrieve context, we first construct a JSON of the specification with the table of contents as the keys and the corresponding sections as values. This is available in our GitHub [21]. For each prompt, we include the corresponding section in the prompt using the JSON. We believe that the JSON would also be valuable to the OpenACC community for projects like this.

To perform similarity search retrieval, we create a vector store containing the specification text. First, we split the text into chunks of 1000 characters, with an overlap of 100 characters - this would give some continuity between chunks. We then create embeddings of the chunks that represent their meaning [50]. Next, we perform a vector store similarity search for each prompt to retrieve relevant information from the specification to the prompt. The retrieved information is then wrapped into the prompt, shown in Figure 2. The necessity of context

```

1 {"prompt": "<prompt text>", "completion": "<ideal generated test>"}
2 {"prompt": "<prompt text>", "completion": "<ideal generated test>"}

```

Figure 3: Basic fine-tuning format, a prompt-completion pair dataset

LLMs	Tests generated	Time Taken	GPUs used	Model Parameter
GPT-3.5-Turbo fine-tuned	351	~4 hours	Unknown	174B
GPT-4	351	~4 hours	Unknown	Unknown
Meta’s Codellama-34B-Instruct	351	~5 hours	4 V100s	34B
Phind-Codellama-34B-v2	351	~5 hours	4 V100s	34B

Table 2: Stage 2 details, including the number of tests generated, inference time, hardware, and model parameters

retrieval arises because the context window of most LLMs is shorter than the length of the entire specification. The whole specification can not fit into the prompt, otherwise an alternative approach would be to include the entire specification in the prompt for each test.

Some LLMs offer a context window that would fit the entire specification, but by providing only relevant information reduces the amount of data it must filter. For example, Meta AI trains all Codellama models with a context window of up to 100k tokens, which is longer than the specification. This means we are able to fit the entire text into the prompt instead of performing RAG. We did so with Codellama-34b-Instruct using Meta AI’s published inference code [51] which only supports parallelization of four GPUs. We observed an out-of-memory issue. Had Meta supported more than four GPUs, we would have overcome this challenge. We are not aware if Meta plans to support more than four GPUs in the near future. We also tried Codellama using Huggingface transformers API [26], which supports parallelism with more than four GPUs, however we observed incorrect output so we tabled it for near-future work.

4.4. One-shot vs Zero-shot Prompting

We compare the performances of zero-shot prompting vs. one-shot prompting by creating two set of prompts, one for each. To construct zero-shot prompts, we do not provide any example of validating OpenACC compiler implementation, however we choose to include a code template to help standardize the generated tests across testsuites. For the one-shot prompts, we provide within each prompt an example prompt and a correct manually written OpenACC V&V test from the OpenACC V&V testsuite [19]. We run both sets of prompts with all selected LLMs. Additionally, we evaluate the performance of one-shot prompting with RAG.

4.5. Expressive Prompt

Besides training, the prompt is the only thing that a LLM has to use to produce the desired output. The prompt is highly influential on the output of the LLM. Even the number of words, choice of words (often termed as word-choice) impacts the quality of the output. In our work we compare the output from a simple prompt with one sentence, versus a detailed prompt with various requirements listed for the task. To this end, we

compare the output between the methods that create the most passing tests without an expressive prompt, to the same methods using an expressive prompt, shown in Figure 2. Improved output with expressive prompt using RAG with code template is shown in Figure 4 (in contrast with a simple prompt and output, shown in Figure 1).

4.6. Dataset and Fine-tuning

We highlight the steps taken to construct a fine-tuning dataset using our manually created OpenACC V&V testsuite and perform fine-tuning of LLMs to improve quality of validation test generation for OpenACC. Each manually written test focuses on one feature, so we create a prompt for each test. Each prompt in the fine-tuning datasets requests a validation test for OpenACC implementations of the feature being tested. The format of LLM fine-tuning datasets is typically a set of prompts and desired responses, as shown in Figure 3.

We create two datasets, a small one comprising of 50 C, C++, and Fortran test examples and a larger dataset comprising of 1335 examples. One-shot examples and templates are not included in the fine-tuning dataset because the goal of fine-tuning is to teach the LLM the desired behaviour without providing instruction in the prompt.

Throughout fine-tuning development, we evaluate the LLM output to ensure that it is tailored to OpenACC and producing relevant code. The foundation models we select for fine-tuning in this research are GPT-3.5-Turbo and Codellama-34b-Instruct. GPT-4 is not available for fine-tuning at this time. We fine-tune both GPT-3.5-Turbo and Codellama-34b-Instruct for 10 epochs. For Codellama-34b-Instruct we encountered out of memory issues and subsequently performed parameter efficient fine-tuning. We report results for GPT-3.5-Turbo in Section 5 however not for Codellama-34b-Instruct as it did not produce coherent code after parameter efficient fine-tuning.

4.7. Errors

Before we talk about the process of development, let us categorize the different types of errors. There is a possibility of three types of errors:

- A parsing error occurs when the generated output is incomplete, and no end to the code block is signified. This occurs when the LLM gets stuck in a infinite loop of

generation, instead of generating an end to the test. The script we use to parse the output looks for an end to the code, through back-ticks or a return statement - a parsing error occurs when neither appears.

- A test is labeled compile error if it does not successfully compile
- A test is labeled runtime error if it fails during execution or returns nonzero,
- A test is labeled pass if it returns 0.

The prompts and fine-tuning methods we implement are designed to generate tests that return a value of 0 if the feature is

working correctly and a non-zero value otherwise. The output of LLMs is typically stochastic, so it cannot be guaranteed that the tests that are generated indeed return 0 only if the feature is working correctly. It may be the case that the LLM writes a test that returns 0 regardless, so the tests require further evaluation.

4.8. Stages

Here we further explain the development process. We have broken down the process into three stages for easier comprehension. We further explain what entails in each of these stages.

Stage 1: First, we use each selected LLM and all 5 methods described in Section 4.2 to generate 95 tests. These are only written in C. These tests cover every OpenACC features listed in chapters 2 and 3 of the specification. We record the results of these generated tests against one OpenACC compiler.

We do not create permutations of tests in this stage, such as creating a test for each clause for each compute construct, i.e. parallel, serial, and kernels. The script we use to compile and run the generated tests labels each test as either parsing error, compile error, run-time fail, pass (definitions can be found in the previous section) We define these categories as the potential outcomes of Stage 1 evaluation.

While we do not collect statistics on the analysis of the tests at this stage, we manually assess the generated tests and consider the results of the runtime scripts in selecting the best approach for Stage 2 which entails further refining our development process.

Stage 2: Based on what we observe in Stage 1, we drop the following methods: code template, RAG with code template, one-shot example, RAG with one-shot example, retaining the expressive prompt using RAG with code template in this stage. For each LLM, we generate 351 tests for every OpenACC feature in C, C++, and Fortran listed in Chapter 2 and 3 of the specification. Additionally, we generate permutations of the compute construct clause tests for each compute construct, i.e. parallel, serial, and kernels, to increase coverage of the specification. Again we record the results of running each generated test against an OpenACC compiler, and select the LLM with the largest number of passing tests for manual analysis in Stage 3. Table 2 lists the full set up for this stage.

Stage 3: Here, we analyze the output of the LLM and method that produced the most passing tests. Through this analysis, we aim to determine where the LLMs and methods fall short in generating correct tests and consider how the quality of test output may be improved. To this end, we manually analyze a representative subset of both passing and failing tests to determine whether the OpenACC implementations are correct. We present the analysis in Section 5.7.

These three stages have been enlightening in a way as the have provided tremendous insights for determining how we can improve the overall development process. We discuss this further in the Section 6.

5. Results

In this section, we summarize the results from our evaluation of LLMs and their capabilities in OpenACC test genera-

```

1 #include "acc_testsuite.h"
2
3 #ifndef T1
4 /* T1: acc delete, V:2.0-2.7 */
5 int test1() {
6     int err = 0;
7     int* h_data;
8     size_t bytes = sizeof(int);
9
10    h_data = (int*)malloc(bytes);
11    *h_data = 15;
12
13    /* Place data on device */
14    acc_copyin((void*)h_data, bytes);
15
16    /* Delete data from device */
17    acc_delete((void*)h_data, bytes);
18
19    /* Data should now not be present in the
20     device */
21    if (acc_is_present((void*)h_data, bytes)) {
22        err = 1;
23    }
24
25    free(h_data);
26
27    return err;
28 }
29 #endif
30
31 int main() {
32     int failcode = 0;
33     int failed;
34
35     #ifndef T1
36         failed = 0;
37         for (int x = 0; x < NUM_TEST_CALLS; ++x) {
38             failed = failed + test1();
39         }
40         if (failed != 0) {
41             failcode = failcode + (1 << 0);
42         }
43     #endif
44
45     return failcode;
46 }

```

Figure 4: Generated test with expressive prompt using RAG with code template. This is a passing test, targeting the correct directive, and using the desired format for the testsuite infrastructure and is generated based on context from the specification.

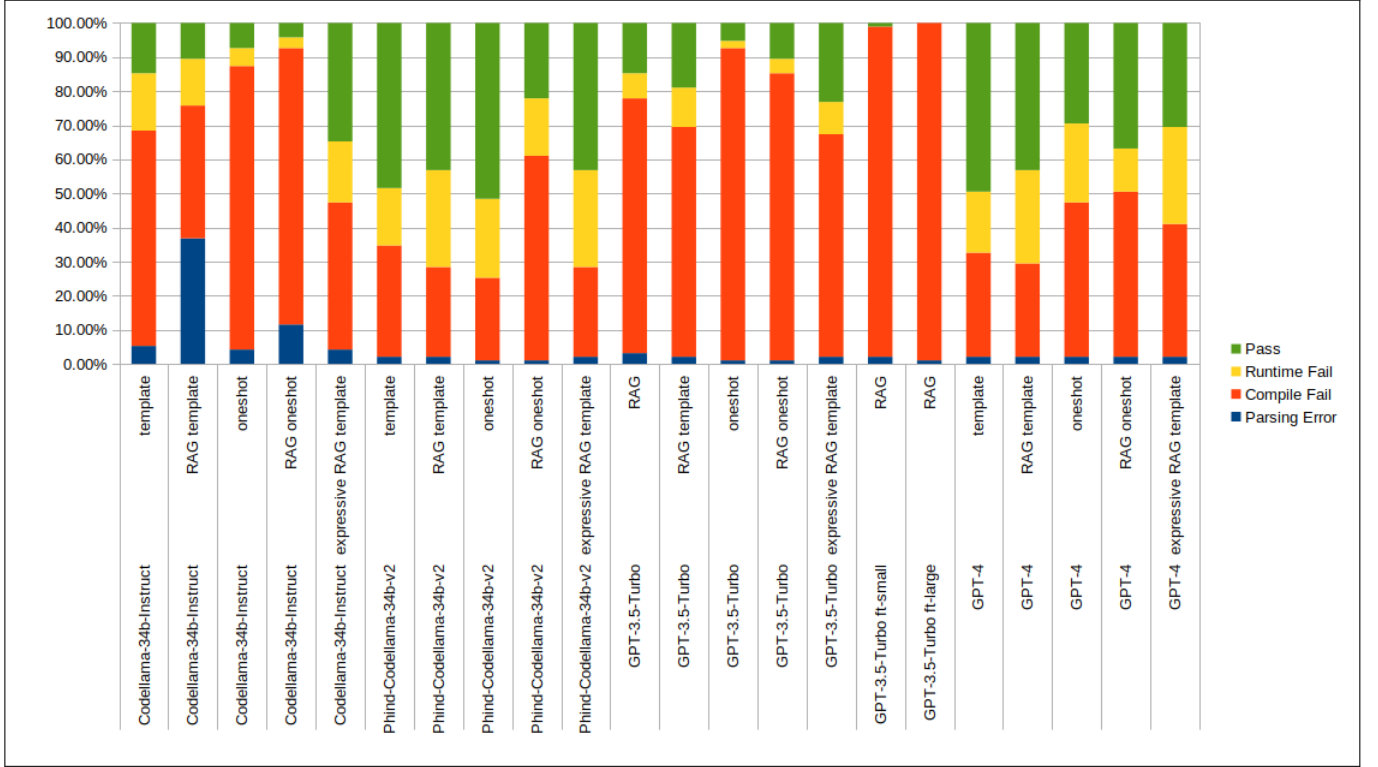


Figure 5: Stage 1 results displaying runtime results for all selected LLMs and methods. We see that Phind-Codellama-34B-v2 and GPT-4 produce the most passing results, using expressive prompt, RAG, and a template.

tion.

5.1. LLMs

To ensure a consistent programming environment to evaluate the performance of selected LLMs and methods, we utilize Anaconda to create a virtual environment and install necessary packages [52]. To locally host open-source LLMs, we use the Llama library for Codellama-34b-Instruct with published inference code [51] and Huggingface transformers [26] for Phind-Codellama-v2 [48]. We use Langchain [53] for text-splitting as a preprocessing step for input into the vector database. We use scikit-learn vector store [54] with Huggingface embeddings through Langchain to create the vector database. For GPT-3.5-Turbo and GPT-4 we use the OpenAI API for all inference and fine-tuning [55]. For fine-tuning open-sourced models, we use a Transfer Reinforcement Learning supervised fine-tuning module [34]. To evaluate the generated tests, we create a python script to compile and run tests using NVIDIA-HPC-SDK 23.5, and record the results [56].

5.2. Experimental Setup

All inference and fine-tuning for Codellama-34b-Instruct and Phind-Codellama-34b-v2 were performed using the National Energy Research Scientific Computing Center’s (NERSC) Perlmutter [57], an HPE Cray EX supercomputer equipped with AMD EPYC CPUs and NVIDIA A100 GPUs and an NSF-sponsored HPC Cluster in UDEL, Darwin [58] equipped with AMD EPYC and Intel Platinum CPU nodes and NVIDIA V100 GPUs and one AMD MI100 GPU.

5.3. Codellama-34b-Instruct

We first generate a testsuite in C using Codellama-34b-Instruct using the prompt methods listed in Section 4.2. We run each generated testsuite against an OpenACC compiler using a script to capture the number of compile errors, runtime errors, and passing tests. These results are displayed in Figure 5.

Codellama-34b-Instruct produced the least number of compile errors using the expressive prompt using RAG with a code template. This is not necessarily indicative of best performance, as the goal of the compiler validation testsuite is not to achieve no errors, but rather to detect existing errors in implementations. Thus, errors are expected to be acceptable outcomes of the testsuite.

In Stage 2, as described in Section 4.8, using the expressive prompt with RAG and template, we generate a testsuite in C, C++, and FORTRAN. As indicated earlier in the same section that there are 351 tests that could be generated out of the two chapters of the specification, 43/351 produce passing results by checking for return code "zero". The bulk of the errors are compile errors.

5.4. Phind-Codellama-34b-v2

From Stage 1 results, we find that Phind-Codellama-34b-v2 produced almost double the number of passing tests using RAG with a code template than RAG with a one-shot example. Additionally, we see that more passing tests are produced without using RAG. Providing the latest version of the OpenACC specification through RAG is essential to producing up-to-date tests,

given that the specification is updated yearly and it is unclear what data relevant to the OpenACC specification LLMs have been pre-trained on. Thus, we retain RAG in Stage 2, selecting again an expressive prompt with a template and RAG, as with Codellama-34b-Instruct.

In Stage 2, Phind-Codellama-34b-v2 produced 95 passing tests out of 351 in the generated testsuite again by checking for return code "zero". Again, the majority of errors are compile errors rather than runtime or parsing errors. We also find that in comparison to Codellama-34b-Instruct, Phind-Codellama-34b-v2 produced less incomplete tests, shown as parsing error labeled in blue in Figure 6.

5.5. GPT-3.5-Turbo

We compare the performance of OpenAI’s GPT-3.5-Turbo using the same five prompting methods as listed in Subsection 4.2. We also evaluate the performance of fine-tuned version of GPT-3.5-Turbo as mentioned in Subsection 4.6. Figure 5 shows that GPT-3.5-Turbo with no fine-tuning using an expressive prompt with RAG and template produced more passing tests than other methods, by checking for return code "zero".

We fine-tune GPT-3.5-Turbo using the small and the large datasets comprising of 50 and 1335 training examples respectively. The generated tests with these models follow the testing format shown in fine-tuning examples without being prompted with a code template or one-shot example, so the results are initially promising. We find in the Stage 1 results that both of these models produce no passing tests, despite producing tests with formatting similar to the test examples in the fine-tuning dataset. We include the model fine-tuned on the large dataset in the final analysis for a comparison with the prompt methods.

In full testsuite generation we evaluate the performance of GPT-3.5-Turbo fine-tuned on 1335 examples. Using only RAG in the prompts, this LLM produced 43 passing tests by checking for return code "zero".

5.6. GPT 4

We evaluate the performance of OpenAI’s GPT-4 in Stage 1 with the 5 prompt methods listed in Section 4.2. GPT-4 produced the most passing tests using only a code template in the prompt, and RAG with a code template produced the second-most. Again, one-shot prompting was outperformed by a code template. In this stage, we see that the expressive prompt using RAG with code template does not produce the most passing tests, but we include it in Stage 2 to provide a consistent comparison among all models.

GPT-4 with expressive prompt using RAG with code template for Stage 2 testsuite generation produced 109 passing tests

GPT-3.5-Turbo	GPT-4	CodeLlama-34B-Instruct	Phind-CodeLlama-34B-v2
43/351	109/351	41/351	95/351

Table 3: Stage 2 results displaying Pass/Fail results for each generated testsuite

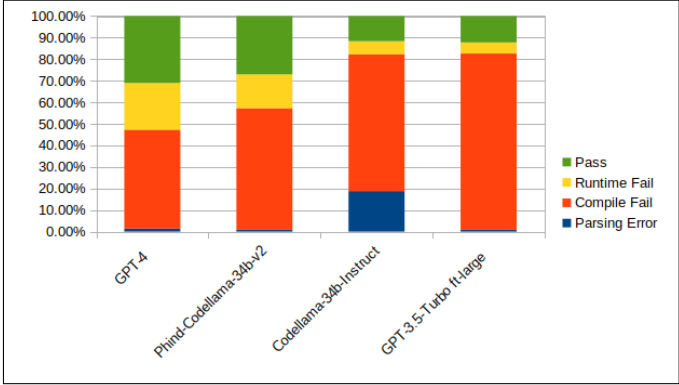


Figure 6: Stage 2 results using expressive prompt with RAG and test template with all selected LLMs. We note that GPT-4 produced the most passing tests, though this is not necessarily indicative of best performance.

out of 351, the most out of any setup tested here. It also did not succumb to many parsing errors, and had the least number of compile errors. Due to these reasons, we decided to manually analyze a representative set of 50 of the generated tests from GPT-4 in a final analysis stage to determine whether the generated tests are implementing accurate testing logic and OpenACC annotations.

The results are tabulated in Table 3.

5.7. GPT-4 Output Analysis

To systematically analyze the tests generated by GPT-4, we manually evaluate a representative subset of 25 passing tests and 25 failing tests. We find that 87.5% (22 tests) of the evaluated passing tests properly implement an OpenACC validation test (green), while 12.5% (3 tests) do not and produce a false passing result (red), shown in Figure 7. A false pass (red) means that the test does not properly test an OpenACC implementation, but still returns 0. For example, a test could be created for the private clause, but the LLM may not actually use the private clause, and still return 0. To evaluate why tests are failing, we consider two cases: incorrect implementation of OpenACC (yellow), or failure despite a correct OpenACC implementation (blue). Incorrect implementation of OpenACC code means that a feature of OpenACC is used incorrectly, thus causing a failure through compile error, runtime error, or a non-zero return value. A failure despite a correct OpenACC implementation means that the OpenACC logic could be correct, but there is an error in the code somewhere else. This could be a syntax error, use of an undefined function, incorrect testing logic, or an unimplemented feature or anything else that may cause a compile error, runtime error, or non-zero return value. An example of such a scenario could be a test that randomly fills an array with numbers, then squares all elements within an OpenACC construct, but then forgets to square the elements of the expected output.

Throughout this analysis, we find that the majority of passing tests are correctly implementing OpenACC (green). The presence of false passing tests (red) reaffirms that human evaluation is necessary for this task. Complete automation is not yet plausible with the techniques described here. In failing tests

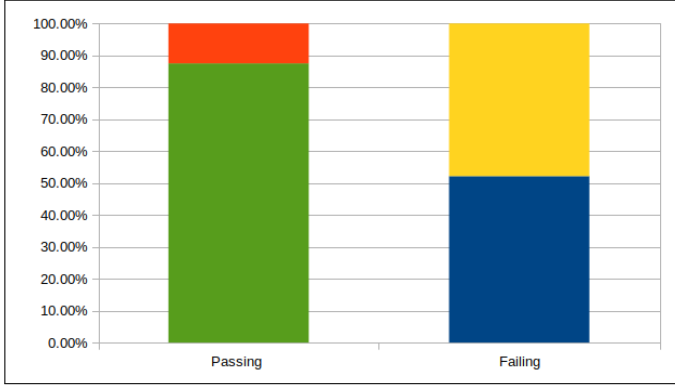


Figure 7: Stage 3 results displaying the analysis of a representative subset of generated tests by GPT-4. Green - True pass. Red - False Pass. Blue - Fails with issues in base language or compiler implementation. Yellow - Fails with incorrect OpenACC usage. The analysis shows that most passing tests are correct tests, whereas failing tests occur due to various reasons.

(blue and yellow), we find a variety of errors consisting of a somewhat even distribution of errors due to incorrect OpenACC logic and other causes.

5.8. Results Summary

Drawing a summary out of the above discussed results and tying them to our contribution list from Section 1.1, we infer the following:

- Out of the five prompt sets evaluated, and fine-tuning with GPT-3.5-Turbo, we find the best performance with an expressive prompt using RAG and a test template.
- For this study the fine-tuning dataset used to finetune GPT-3.5-Turbo enabled the LLM to produce correct format of test without a one-shot example or code template, as expected. However, the test quality still lacked indicated by low passing rate.
- Out of the four models evaluated in Stage 2, we find that GPT-4 produces the most passing tests
- In the final results analysis, we find that most passing tests implement correct OpenACC logic, while failing tests fail for a variety of reasons.

6. Discussion

By automating the process of generating functional test cases, we are enabling the developers to not spend their time to write these tests, instead spend their time on designing regression and corner case tests - tests that we have not yet fully explored LLMs for nor are we aware at this point how suitable LLMs would be for those cases. Moreover, validation testsuite generation not a one and done task. The specification of programming models constantly evolves. So a validation testsuite must be maintained by a team of developers to be consistent with the latest version of the specification. Until recently, natural-language processing techniques were not advanced enough to

comprehend a long, complex programming model specification and generate code to validate compiler implementations of the specification. With LLMs, its much more viable. When the specification is updated, an LLM can be used again to generate an up-to-date testsuite.

We found that our initial expectations of the relative performance of the selected models, based on benchmarks and related work, were confirmed by our evaluation. We did not expect this research in its first iteration, to necessarily implement a perfect testsuite generation pipeline, but rather provide insight to the capabilities of current models and areas for improvement and motivation to adapt ideas to several other testsuite projects. The performance of the selected LLMs on the HumanEval benchmark trends similarly to the pass rates and performance we found in OpenACC testsuite generation. Though this is a reassuring result, we plan to create a high-performance computing code generation benchmark especially with a focus on testing for LLMs in the future.

To discuss the results further, first, we find that the use of a simple code template rather than a one-shot example of test implementation produced more passing tests. The difference between these methods is that the template does not provide any example OpenACC code or testing logic. This is interesting to observe because the full example provides more context.

A potential cause in this result is the increase in length of the input creating more complexity. It introduces potentially unrelated OpenACC features into the prompt, and increases the number of tokens in the input. The prompt all-together is long - when using RAG, we include in each prompt the corresponding section of the specification for the feature being tested. The prompt itself is expressive, and the one-shot example is a full test. All of this adds up to a long prompt, and additional complexity from a full one-shot example seems to negatively impact the test quality in this research.

Using a simple template reduces the input length and complexity, allowing LLM to focus on the feature at hand, and not the feature in the example. The use of an expressive prompt, detailing the goal of the test and the desired behaviour, generally improved performance, despite the increase in input length. We believe that the expressive prompt includes relevant context to the task, while the one-shot example provided information about irrelevant OpenACC features, thus confusing the LLM leading to hallucinations.

Based on the manually created OpenACC testsuite results, we initially expected higher pass rates for the testsuites generated in this work. The OpenACC V&V testsuite produces 81.4% passing tests (1087/1335) while the highest passing rate from Stage 2 using GPT-4 is 31% (109/351). An important distinction here is the number of tests. The manually written testsuite consists of 1335 tests, while the full test generated in this work consist of 351 tests. The manually written tests consider more permutations of features than this work. Thus, number of tests for each feature is not consistent between the tests generated in this work and the manually written suite, so the passing rate is not a fair comparison. Nevertheless, we find through evaluation that many tests that fail are indeed due to issues in LLM, rather than incorrect implementation of OpenACC fea-

tures by compilers.

We discuss GPT-4 output analysis in Section 5.7 and the different scenarios where tests fail. We find that the incorrect use of OpenACC features often occurred not due to incorrect use of the feature being tested, but rather due to improper use of other OpenACC features within the test. For example, the generated test might target the copyin clause, but the LLM may implement another clause in the test, and use it incorrectly. A potential solution to this source of error is to implement a RAG method that includes the sections from the specification for all OpenACC features used in each test, not only the single feature being tested. This solution would involve two steps. First, planning of a test. Then, retrieval of all relevant sections, and second, the generation of the test.

Another method that we suggest could improve the performance of this work greatly is to give the LLM multiple attempts at each test. This could involve simultaneous generation of multiple tests, compilation of each and running of each, analysis of the output, and selection of the best test based on the output. Within LLM benchmarking, this method is known as the pass@k metric, where LLMs are given more than one attempt at completing the task of the benchmark, e.g. code generation.

7. Conclusion

The purpose of this work is to apply LLMs to generation of OpenACC test cases which in turn would be used to validate and verify compiler implementations of the OpenACC specification. As the specification is updated, the pipeline can be run again to generate a new suite of tests that verify the newly added features. Moreover, the improvement of techniques in optimizing large language model performance will likely improve the overall performance of test case generation. This way we aim to "shift" where the developers use their valuable time and not "replace" their efforts. This project requires human oversight but at the same time also relieves them from writing tests that LLMs can do a good job at.

This project is meant as a new application of the emerging LLMs in high performance computing. As an immediate and on-going work we have started to look into optimizing our methods and exploring the adaptability to the OpenMP testsuite. As near future steps we will work with developers of Kokkos, RAJA, Chapel, SYCL among others and explore the usability and expandability of the approach based on their inputs and needs.

The harsh truth about the current state of LLMs is that they can generate false and misleading output at times, despite various efforts to discourage this behavior and we know that LLMs hallucinate. We emphasize the need to exercise caution and conduct human evaluation on all outputs that are used in production. The accuracy of these methods is likely to improve as the community continues to learn and make adjustments.

8. Acknowledgments

This research was supported by the National Science Foundation (NSF) under grant no. 1919839, in part through the use

of DARWIN computing system: DARWIN – A Resource for Computational and Data-intensive Research at the University of Delaware. This material is also based upon work supported by NSF under grant no. 1814609.

This research also used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

This research also used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231 using NERSC award. We are grateful to OpenACC for their support on this project.

References

- [1] OpenAI, GPT-4 technical report (2023). arXiv:2303.08774.
- [2] Anthropic, Claude 2 (2023). URL <https://www.anthropic.com/index/claude-2>
- [3] GitHub, Github copilot (2023). URL <https://github.com/features/preview/copilot-x>
- [4] codeium, Codeium · free ai code completion & chat (2022). URL <https://codeium.com/>
- [5] OpenMP, The openmp api specification for parallel programming (2023). URL <https://www.openmp.org/>
- [6] OpenACC, Openacc specification (2022). URL <https://www.openacc.org/specification>
- [7] L. Zhong, Z. Wang, A study on robustness and reliability of large language model code generation (2023). arXiv:2308.10335.
- [8] Gcc supporting openacc model, <https://gcc.gnu.org/wiki/OpenACC>.
- [9] J. E. Denny, S. Lee, J. S. Vetter, Clacc: Translating openacc to openmp in clang, in: 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC), IEEE, 2018, pp. 18–29.
- [10] NVIDIA, CUDA SDK Code Samples, <http://developer.nvidia.com/cuda-cc-sdk-code-samples>, accessed: 2017-02-03.
- [11] OpenCL, OpenCL, <https://www.khronos.org/>.
- [12] A. Alpay, V. Heuveline, Sycl beyond opencl: The architecture, current state and future direction of hipsycl, in: Proceedings of the International Workshop on OpenCL, 2020, pp. 1–1.
- [13] W. Sawyer, G. Zaengl, L. Linardakis, Towards a multi-node openacc implementation of the icon model, in: EGU General Assembly Conference Abstracts, 2014, p. 15276.
- [14] B. Kim, K. S. Yoon, H.-J. Kim, Gpu-accelerated laplace equation model development based on cuda fortran, Water 13 (23) (2021) 3435.
- [15] X. Lapillonne, O. Fuhrer, Using compiler directives to port large scientific applications to gpus: An example from atmospheric science, Parallel Processing Letters 24 (01) (2014) 1450003.
- [16] S. Sathe, Accelerating the ansys fluent r18.0 radiation solver with openacc (2016).
- [17] R. Searles, S. Chandrasekaran, W. Joubert, O. Hernandez, Mpi+ openacc: Accelerating radiation transport mini-application, minisweep, on heterogeneous systems, Computer Physics Communications 236 (2019) 176–187.
- [18] M. Stack, P. Macklin, R. Searles, S. Chandrasekaran, Openacc acceleration of an agent-based biological simulation framework, Computing in Science & Engineering 24 (5) (2022) 53–63.
- [19] A. Jarmusch, A. Liu, C. Munley, D. Horta, V. Ravichandran, J. Denny, K. Friedline, S. Chandrasekaran, Analysis of validating and verifying openacc compilers 3.0 and above, in: 2022 Workshop on Accelerator Programming Using Directives (WACCPD), 2022, pp. 1–10. doi: 10.1109/WACCPD56842.2022.00006.
- [20] T. Huber, S. Pophale, N. Baker, M. Carr, N. Rao, J. Reap, K. Holsapple, J. H. Davis, T. Burnus, S. Lee, et al., Ecp solve: Validation and verification testsuite status update and compiler insight for openmp, in: 2022

- IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), IEEE, 2022, pp. 123–135.
- [21] LLMs for testsuite generation, <https://github.com/chrismun/vv11m>.
- [22] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, I. Polosukhin, Attention is all you need (2023). [arXiv:1706.03762](https://arxiv.org/abs/1706.03762).
- [23] Anthropic, Introducing claude (2022). URL <https://www.anthropic.com/index/introducing-claude>
- [24] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, G. Synnaeve, Code llama: Open foundation models for code (2023). [arXiv:2308.12950](https://arxiv.org/abs/2308.12950).
- [25] R. Taori*, I. Gulrajani*, T. Zhang*, Y. Dubois*, X. Li*, C. Guestrin, P. Liang, T. B. Hashimoto, Alpaca: A strong, replicable instruction-following model (2022). URL <https://crfm.stanford.edu/2023/03/13/alpaca.html>
- [26] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, A. M. Rush, Huggingface’s transformers: State-of-the-art natural language processing (2020). [arXiv:1910.03771](https://arxiv.org/abs/1910.03771).
- [27] J. Webster, C. Kit, Tokenization as the initial phase in nlp, 1992, pp. 1106–1110. doi:10.3115/992424.992434.
- [28] A. Tamkin, M. Brundage, J. Clark, D. Ganguli, Understanding the capabilities, limitations, and societal impact of large language models (2021). [arXiv:2102.02503](https://arxiv.org/abs/2102.02503).
- [29] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, D. Amodei, Language models are few-shot learners (2020). [arXiv:2005.14165](https://arxiv.org/abs/2005.14165).
- [30] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. tau Yih, T. Rocktäschel, S. Riedel, D. Kiela, Retrieval-augmented generation for knowledge-intensive nlp tasks (2021). [arXiv:2005.11401](https://arxiv.org/abs/2005.11401).
- [31] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, D. Zhou, Chain-of-thought prompting elicits reasoning in large language models (2023). [arXiv:2201.11903](https://arxiv.org/abs/2201.11903).
- [32] B. Lester, R. Al-Rfou, N. Constant, The power of scale for parameter-efficient prompt tuning (2021). [arXiv:2104.08691](https://arxiv.org/abs/2104.08691).
- [33] J. Wei, M. Bosma, V. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, Q. V. Le, Finetuned language models are zero-shot learners, in: International Conference on Learning Representations, 2022. URL <https://openreview.net/forum?id=gEZrGCozdqR>
- [34] L. von Werra, Y. Belkada, L. Tunstall, E. Beeching, T. Thrush, N. Lambert, S. Huang, Trl: Transformer reinforcement learning, <https://github.com/huggingface/trl> (2020).
- [35] T. Dettmers, A. Pagnoni, A. Holtzman, L. Zettlemoyer, Qlora: Efficient finetuning of quantized llms (2023). [arXiv:2305.14314](https://arxiv.org/abs/2305.14314).
- [36] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, W. Zaremba, Evaluating large language models trained on code (2021). [arXiv:2107.03374](https://arxiv.org/abs/2107.03374).
- [37] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, C. Sutton, Program synthesis with large language models (2021). [arXiv:2108.07732](https://arxiv.org/abs/2108.07732).
- [38] AWS, Introducing amazon codewhisperer, the ml-powered coding companion (2023). URL <https://aws.amazon.com/blogs/machine-learning/introducing-amazon-codewhisperer-the-ml-powered-coding-companion/>
- [39] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umaphathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kurnakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, H. de Vries, Starcoder: may the source be with you! (2023). [arXiv:2305.06161](https://arxiv.org/abs/2305.06161).
- [40] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, D. Jiang, Wizardcoder: Empowering code large language models with evol-instruct (2023). [arXiv:2306.08568](https://arxiv.org/abs/2306.08568).
- [41] L. Chen, P.-H. Lin, T. Vanderbruggen, C. Liao, M. Emani, B. de Supinski, Lm4hpc: Towards effective language model application in high-performance computing (2023). [arXiv:2306.14979](https://arxiv.org/abs/2306.14979).
- [42] T. Kadosh, N. Hasabnis, V. A. Vo, N. Schneider, N. Krien, A. Wasay, N. Ahmed, T. Willke, G. Tamir, Y. Pinter, T. Mattson, G. Oren, Scope is all you need: Transforming llms for hpc code (2023). [arXiv:2308.09440](https://arxiv.org/abs/2308.09440).
- [43] W. Godoy, P. Valero-Lara, K. Teranishi, P. Balaprakash, J. Vetter, Evaluation of openai codex for hpc parallel programming models kernel generation, in: Proceedings of the 52nd International Conference on Parallel Processing Workshops, ICCP Workshops ’23, Association for Computing Machinery, New York, NY, USA, 2023, p. 136–144. doi:10.1145/3605731.3605886. URL <https://doi.org/10.1145/3605731.3605886>
- [44] D. Nichols, A. Marathe, H. Menon, T. Gamblin, A. Bhatele, Modeling parallel programs using large language models, [arXiv preprint arXiv:2306.17281](https://arxiv.org/abs/2306.17281).
- [45] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, W. Chen, Codet: Code generation with generated tests (2022). [arXiv:2207.10397](https://arxiv.org/abs/2207.10397).
- [46] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, N. Sundaresan, Unit test case generation with transformers and focal context, [arXiv \(May 2021\)](https://arxiv.org/abs/2105.04000). URL <https://www.microsoft.com/en-us/research/publication/unit-test-case-generation-with-transformers-and-focal-context/>
- [47] M. Schäfer, S. Nadi, A. Eghbali, F. Tip, An empirical evaluation of using large language models for automated unit test generation (2023). [arXiv:2302.06527](https://arxiv.org/abs/2302.06527).
- [48] Phind, Phind/phind-codellama-34b-v2 · hugging face (2023). URL <https://huggingface.co/Phind/Phind-CodeLlama-34B-v2>
- [49] N. McKenna, T. Li, L. Cheng, M. J. Hosseini, M. Johnson, M. Steedman, Sources of hallucination by large language models on inference tasks (2023). [arXiv:2305.14552](https://arxiv.org/abs/2305.14552).
- [50] M. Mars, From word embeddings to pre-trained language models: A state-of-the-art walkthrough, Applied Sciences 12 (17). doi:10.3390/app12178805. URL <https://www.mdpi.com/2076-3417/12/17/8805>
- [51] facebookresearch, codellama (2023). URL <https://github.com/facebookresearch/codellama>
- [52] Anaconda software distribution (2020). URL <https://docs.anaconda.com/>
- [53] Langchain, Langchain (2023). URL <https://www.langchain.com/>
- [54] LangChain, scikit-learn (2022). URL <https://python.langchain.com/docs/integrations/vectorstores/sklearn>
- [55] OpenAI, Openai api (2023). URL <https://openai.com/blog/openai-api>
- [56] NVIDIA, Nvidia hpc sdk (2023). URL <https://developer.nvidia.com/hpc-sdk>
- [57] NERSC, Using perlmutter (2023). URL <https://docs.nersc.gov/systems/perlmutter/>

[58] U. of Delaware, Darwin (2023).
URL <https://dsi.udel.edu/core/computational-resources/darwin/>