Using a Large Language Model as a Building Block to Generate Usable Validation and Verification Suite for OpenMP

SWAROOP POPHALE, Oak Ridge National Laboratory, United States
WAEL ELWASIF, Oak Ridge National Laboratory, United States
DAVID E. BERNHOLDT, Oak Ridge National Laboratory, United States

In the HPC area, both hardware and software move quickly. Often new hardware is developed and deployed, the corresponding software stack, including compilers and other tools, are under active development while leading edge software developers are working to port and tune their applications, all at the same time. While the software ecosystem is in flux, one of the key challenges for users is obtaining insight into the state of implementation of key features in the programming languages and models their applications are using – whether they have been implemented, and whether the implementation conforms to the specification, especially for newly implemented features (less tested by widespread use). OpenMP is one of the most prominent shared memory programming models used for on-node programming in HPC. With the shift towards accelerators (such as GPUs and FPGAs) and heterogeneous programming OpenMP features are getting more complex. It is natural to ask whether generative AI approaches, and large language models (LLMs) in particular, can help in producing validation and verification test suites to allow users better and faster insights into the availability and correctness of OpenMP features of interest. In this work, we explore the use of ChatGPT-4 to generate a suite of tests for OpenMP features. We have chosen a set of directives and clauses, a total of 78 combinations, which first appeared in OpenMP 3.0 (released in May 2008) but are also relevant for accelerators. We prompted ChatGPT to generate tests in the C and Fortran languages, for both host (CPU) and device (accelerator). On the Summit super-computer using the GNU implementation, we found that, of the 78 generated tests 67 C tests and 43 Fortran tests compiled successfully and fewer than those executed to completion. On further analysis we show that not all generated tests are valid. We document the process, results, and provide detailed analysis regarding the quality of tests generated. With the aim of providing input to a production quality validation and verification suite, we manually implement the corrections required to make the tests valid according to the current OpenMP specification. We quantify this effort as small, medium, or large, and record the lines of code changed to correct the invalid tests. With the corrected tests we validate recent implementations from HPE, AMD, and GNU on the Frontier supercomputer. Our experiment and subsequent analysis show that although LLMs are capable of producing HPC specific codes, they are limited by their understanding of the deeper semantics and restrictions of a programming models such as OpenMP. Unsurprisingly more commonly used features have better support, while some OpenMP 3.0 directives such as *sections* and *tasking* are not universally supported on accelerators. We demonstrate that successful compilation and execution to completion are inadequate metrics for evaluating generated code and that, at this time, commodity

Authors' addresses: Swaroop Pophale, Oak Ridge National Laboratory, Oak Ridge, Tennessee, United States, pophaless@ornl.gov; Wael Elwasif, Oak Ridge National Laboratory, Oak Ridge, Tennessee, United States, elwasifwr@ornl.gov; David E. Bernholdt, Oak Ridge National Laboratory, Oak Ridge, Tennessee, United States, bernholdtde@ornl.gov@ornl.gov.

Manuscript submitted to ACM

LLMs require expert intervention for code verification. This points to gaps in the training data that is currently available for HPC. We demonstrate that with "small" effort 37% of generated invalid C tests and 63% of generated invalid Fortran tests could be corrected. This improves productivity of test generation as we circumvent writing from scratch and the common programming errors associated with it.

CCS Concepts: • **Software and its engineering → Software prototyping**.

ACM Reference Format:
Swaroop Pophale, Wael Elwasif, and David E. Bernholdt. 2024. Using a Large Language Model as a Building Block to Generate Usable Validation and Verification Suite for OpenMP . In *Proceedings of HPC Asia 2025*. ACM, New York, NY, USA, 17 pages.

1 INTRODUCTION

OpenMP [27] is one of the most popular choices for portable shared memory programming in high performance computing (HPC). Frequently used with distributed programming libraries like MPI [19] or partitioned global address space (PGAS) programming models, OpenMP can distribute work, synchronization, transfer data, and recently, offload computations to other compute devices. The first official specification of OpenMP was released in 1997 and only targeted Fortran. A separate C/C++ specification followed later in October 1998. OpenMP specification 2.5 was released in 2005 as the first version that combines C/C++ and Fortran in the same document. The early versions of the OpenMP specification exclusively targeted a shared memory fork/join parallel programming model, where a group of threads form a *parallel region* that can be used to share work (e.g. loop iterations) across available hardware execution units (e.g. cores in a multi-core CPU). The execution model was extended to support explicit tasks in version 3.0 (released in 2008), with support for task dependency (allowing the formation of complex task graphs) added in version 4.0 [25], released in 2013. Version 4.0 also included the earliest OpenMP support for offloaded execution on separate devices (accelerators), as a result of the emerging use of GPUs in HPC. Support for offloading was greatly enhanced in OpenMP version 4.5 (released in 2015) [26], and the more recent 5.X releases, culminating in the current release (OpenMP 5.2), released in 2021 [27].

As OpenMP's feature set increases so does the complexity of implementations and the need to verify their correctness. Few open-source validation suites exist that are not tied to a particular compiler or development framework [6, 13, 18]. A few that are or have been popular in the HPC community include: the OpenMP Validation and Verification Suite [8, 22], OvO [2], the Barcelona OpenMP Tasks Suite (BOTS) [9], and the OpenMP 3.0 test-suite [3, 4]. Apart from the first two, most of these test suites are not maintained nor updated consistently. Another gap in testing we noticed was that after the device offload model was introduced in OpenMP Specification 4.0 [25] no open-source test suite has made the effort to cover older features in the context of device offload. The closest open-source effort is the OpenMP Validation and Verification Suite [22], which was originally developed under the Exascale Computation Project (ECP) SOLLVE project, which also targets newer directives introduced in OpenMP (4.5 and beyond). Although such efforts are invaluable for the community, vendors, and major computing facilities, the gap in testing older OpenMP features with newer offloading concepts opens a whole new avenue of potentially harmful undiscovered bugs that can hinder application development and lead to expensive discovery and debug cycles.

Large-language models (LLMs) have proved very useful in generating contextually relevant text and natural language understanding. Recently LLMs have been adapted to assist in software development by generating code snippets, explaining existing code, and even providing debugging help [12, 23, 29, 36]. This is especially useful for work like

automating mundane coding tasks or interactively providing suggestions. Many researchers are currently exploring the limits and ways of harnessing their immense potential for code generation for HPC.

In this paper, we explore the use of a commodity LLM (ChatGPT-4) to generate tests for OpenMP features as a means of augmenting and filling in gaps in the available vendor- and implementation-agnostic test suites. We look particularly at tests for OpenMP 3.0 features in conjunction with offloading, which have been valid since OpenMP 4.0. These cases were chosen expecting that more code representing OpenMP 3.0 and 4.0 features and capabilities would have been present in the LLM training data than more recent OpenMP features. The paper achieves the following objectives:

- Demonstrates step-by-step process of generating OpenMP accelerator tests using LLMs
- Quality analysis of generated code using OpenMP specification as our guide, including manual interventions needed to get to usable code
- Provide insights into changes and improvements required to the generated production-grade validation and verification suite for OpenMP 3.0 features in conjunction with offloading.
- Evaluation of multiple OpenMP implementations using the corrected tests.

2 BACKGROUND

2.1 OpenMP

OpenMP is a widely used API that supports multi-platform shared memory multiprocessing programming for C, C++, and Fortran languages. Through the use of straightforward compiler directives and runtime functions, it provides a simple and flexible interface for developing parallel applications on multi-core processors. OpenMP provides efficient utilization of multiple CPU cores and, more recently, "accelerator" devices such as GPUs and FPGAs. It includes directives for work-sharing and synchronization, and environment variables, which help optimize performance and scalability on a variety of platforms. The latest official OpenMP 5.2 specification [27] released in November 2021 added important features such as metadirectives, assumption directives, nothing directive, error directives, allocators construct, and dispatch construct. This work focuses on features of the OpenMP 3.0 [24] specification (released in 2008), which were extended to support accelerator devices in OpenMP 4.0 [25] (released in 2013).

2.2 LLM

There are many publicly available LLMs that specialize in specific functionality. To list a few, there is the GPT series by OpenAI [23], Large Language Model Meta AI (LLaMA) by Meta [31], Bidirectional Encoder Representations from Transformers (BERT) [7] and Text-To-Text Transfer Transformer (T5) [30] by Google, GPT-Neo [10] and GPT-J [33] by EleutherAI, BLOOM by BigScience [35], Claude by Anthropic [1], Command R by Cohere [5], and Alpaca by Stanford University [32]. Each of these LLMs was developed by training specialized neural networks using pre-processed and cleaned data. We choose ChatGPT-4 for our generation of OpenMP tests, as GPT has demonstrated ease of use, advanced language generation and understanding capabilities, and is known for successfully handling a broad range of tasks. It has also shown better OpenMP code generation in previously published works [20].

2.3 Prompt Engineering

Prompt engineering is the practice of designing, refining and optimizing input keywords to effectively guide an LLM in generating the desired output. With the availability of LLMs researchers began by experimenting with how slight changes in phrasing affected model outputs. As currently practiced, prompt engineering involves crafting specific text

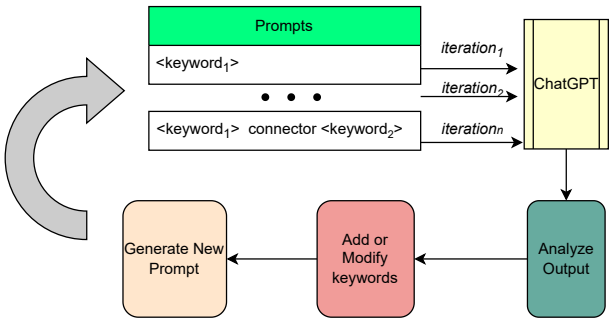


Fig. 1. Prompt Specialization Feedback Loop

inputs or tokens that lead to contextually accurate, relevant, and correct responses from an LLM. Prompt engineering has become a key skill in working with large language models and is of significant consequence. With the capability for interactive refinement, an LLM can successfully achieve more complex tasks by the step-by-step approach of guiding the LLM via domain- and objective-specific prompts.

3 CODE GENERATION

We use the few-shot approach for test generation. The objective was to generate self-validating independent tests for OpenMP 3.0 directives with computation offloaded to device using the `target` directive. Focus was laid on key *tokens*. Since the functionality of the directives does not change from C to C++, we only generated tests for C and Fortran. Our initial approach was a very general prompt using the token *generate* and *offloading tests* for all OpenMP 3.0 features. The response was a python script for plugging in directive and clause combinations to generate a combination of tests. Unfortunately, the script did not account for the different semantics of the directives, restrictions on the clauses, nor did it correctly encode the required data transfers to the device. We had to further specialize our prompts to get meaningful responses.

The iterative design process is shown in Figure 1. We used a feedback mechanism to iteratively refine our prompts. With every iteration the prompts are made more specific based on the response from ChatGPT. The prompts that worked best were explicit, expressive prompts that mentioned the programming language, directive, clause, and the tokens *offloading support* and *testing* with the appropriate propositions joining them.

ChatGPT generated both C and Fortran tests for the same prompt. With the final prompt "`Create an OpenMP 4.5 offloading test for C and Fortran-90 for the <omp directive> directive, with the <omp clause> clause`" with the 19 directives with different clause combinations as shown in Table 1, we were successful in generating 78 distinct tests for OpenMP 3.0 with offloading support.

4 EVALUATION

As discussed in Section 3, 78 distinct tests were generated for a specific directive and clause combination. All tests had the correct header files included; including <omp.h> (even where OpenMP function calls were not used). Our approach was to first try to compile these tests on the Summit supercomputer (POWER9 CPU with NVIDIA V100 GPU) at the

Table 1. List of OpenMP directives and their clauses for which tests were successfully generated. The term *standalone* denotes the directive being used without modifying clauses. Tests for a total of 78 combinations were successfully generated.

OpenMP DIRECTIVE	CLAUSES
atomic	capture, read, update, write
barrier	*standalone*
cancel	parallel, sections, taskgroup
cancellation	*standalone*
critical	name
declare reduction	initializer
declare simd	aligned, linear, simdlen, uniform
flush	*standalone*
for	collapse, private, firstprivate, lastprivate, nowait, ordered, schedule
master	*standalone*
ordered	*standalone*
parallel	copyin, default, firstprivate, if, num_threads, private, shared, reduction
parallel for	collapse, copyin, default, firstprivate, if, nowait, num_threads, ordered, private, reduction, schedule, shared
parallel sections	copyin, default, firstprivate, if, nowait, num_threads, private, shared, reduction
sections	*standalone*, nowait, num_threads, private, firstprivate, lastprivate, reduction, shared
single	copyprivate, firstprivate, nowait, private
task	default, depend, final, firstprivate, if, mergable, priority, private, shared, untied
taskgroup	reduction
taskwait	*standalone*
taskyield	*standalone*

Table 2. Initial assessment of tests generated by ChatGPT using the GNU compilers.

	Language	
Number of	**C**	**Fortran**
Generated tests	78	78
Successfully compiled tests	67 (86%)	43 (54%)
Executed to completion on host	64 (82%)	43 (54%)
Executed to completion on device	60 (78%)	43 (54%)

Oak Ridge Leadership Facility (OLCF). We chose the GNU compiler for this step as it has mature and stable support for this platform, along with good OpenMP feature support for both C and Fortran.

4.1 Quality of Generated Tests

4.1.1 Compile and Execution Time Results. We used the GNU GCC compiler (development version 13.2.1) enabled for device offload on the Summit super-computer for evaluating the generated tests. Out of the box we had 67 and 43 tests compile successfully for C and Fortran respectively as shown in Table 2.

As shown Table 3, the most common reason for compilation failure for C tests were OpenMP specification violations while for Fortran it was language errors. We discuss some interesting cases here which shed light on the limitations of LLMs to grasp programming model and language semantics. For example, for the test for the `parallel for` directive with the `copyin` clause, the LLM generated the code listed in Listing 1. One obvious error is that *shared_var* is used in the `threadprivate` directive before it is declared. This is easily fixed by moving the declaration in line 2 to before

Table 3. High-level classification of errors encountered. Errors occurred at either during compilation or execution. Compile-time errors are attributed to either non-compliance with the OpenMP specification or with the base language specification (C or Fortran). Execution-time errors are due to lack of support for certain offload features, errors in semantics and race conditions in generated code.

Error Category	Error Sub-Category	Language C	Language Fortran
Compile-time	OpenMP non-compliance	10	11
	Base language non-compliance	0	24
Execution-time	Test failure	10	4
	Memory failure	2	39

```
1  #pragma omp threadprivate(shared_var)
2  int shared_var;
3
4  int main() {
5      /* intializarions of variables*/
6
7      // Initialize shared variable and results array
8      shared_var = 42;
9      for (i = 0; i < n; i++) {
10         results[i] = -1;
11     }
12
13     // Offload computation to target device
14     #pragma omp target map(tofrom: results)
15     {
16         #pragma omp parallel for copyin(shared_var)
17         for (i = 0; i < n; i++) {
18             results[i] = shared_var + i;
19         }
20     }
```

Listing 1. Main body of the offlading test for `parallel for` directive with the `copyin` clause generated by ChatGPT

its use in line 1. The more important issue is the clashing semantics of the `copyin` clause and `target` construct. The `copyin` clause is used to copy the value of the `threadprivate` variable *shared_var* from the primary thread to all other threads executing the `parallel` region. According to the OpenMP specification accessing a `threadprivate` variable in the `target` region leads to unspecified behavior. The model could not infer these restrictions and incorrectly used the `threadprivate` variable *shared_var* in the `target` region. The correction in this case is to remove the offloading (line 14) from the mix as it clearly leads to non-compliant code which may not be portable or have reproducible output. Similar errors were observed in other tests that involved the `copyin` clause and all of them were corrected in a similar manner. Another common set of errors in C tests were in the case of the `default` clause. When the `default` clause is used to determine the implicit data-sharing attributes of variables referenced in the construct. With `default(none)` the data-sharing attributes of all the variables referenced in the enclosing construct have to be specified.

The generated Fortran tests exhibited a number of base language errors. Several tests include a declaration of a Fortran variable variable within a `DO` loop, as shown in the listing 2. Although such a declaration is valid in the C language and would be considered a `private` variable according to the OpenMP specification, it is not allowed in Fortran. This error is corrected by moving the declaration of *private_var* to the declarations part at the top of the Fortran test, and explicitly adding the variable *private_var* to the list of items in the clause `private` in addition to the variable *i*.

```
!$omp target map(tofrom: results1, results2)
!$omp parallel sections private(i)
!$omp section
  do i = 1, n
    integer :: private_var
    private_var = (i - 1) * 2
    results1(i) = private_var
  end do
```

Listing 2. Incorrect Fortran variable declaration in test for `sections private` OpenMP construct

```
program offload_test
! ..... ommitted code
  do i = 0, n-1
    expected_result = expected_result + i
  end do

  ! Declare a subroutine to perform a task
  subroutine perform_task(id, shared_var)
    integer, intent(in) :: id
    integer, intent(inout) :: shared_var
    !$omp task shared(shared_var)
    print *, 'Task ', id, ' is executing'
    !$omp atomic
    shared_var = shared_var + id
    !$omp end task
  end subroutine perform_task

```

Listing 3. Incorrect Fortran subroutine definition in generated tests

Table 4. Status of tests after fixing trivial compile-time errors. Percentages are relative to the originally-generated 78 tests. Note that successful execution does not imply anything about the *validity* of the test.

	Language	
Number of tests	**C**	**Fortran**
Compiled and executed to completion on **host**	73 (94%)	60 (77%)
Compiled and executed to completion on **device**	72 (92%)	60 (77%)

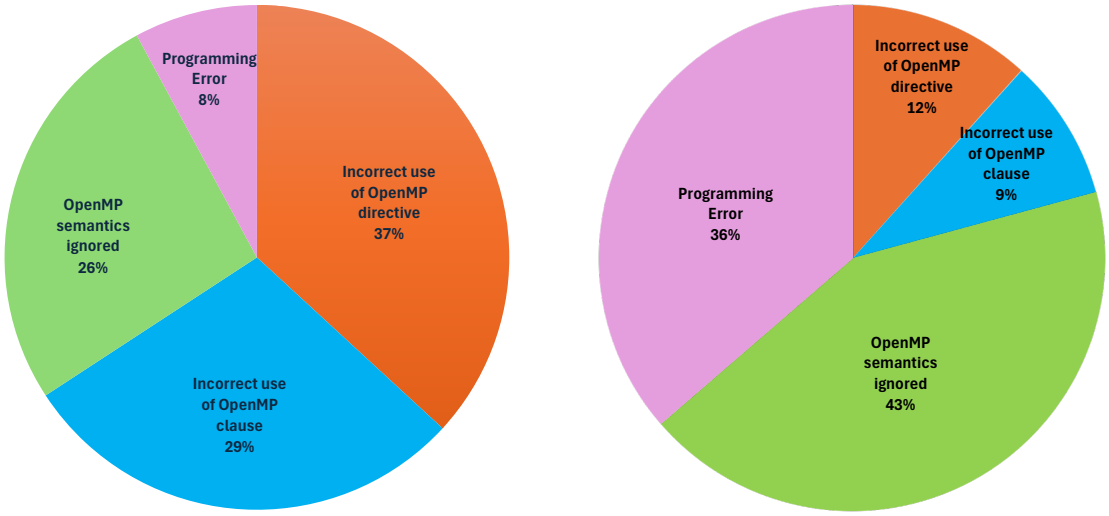
Another common error in generated Fortran tests involved the placement of subroutines inside the main program scope as shown in Listing 3. Such a construction is valid in the C language, where Fortran requires the subroutine to be either declared outside the main `program` scope, or inside the `contains` block of the program or module.

Table 4 summarizes the results obtained after trivial compile-time errors in the generated tests were corrected. More than 92% of C-language and and 77% of Fortran tests were able to compile and run after minor fixes. Note that successful execution does not imply that the tests generated are valid OpenMP codes.

4.1.2 Invalid Tests. Not all of the generated tests, even those that compiled and executed to completion, were valid OpenMP tests. Table 5 gives the distribution of valid vs. invalid tests generated, based on inspection of each test. While most related works stop at this stage declaring one or another LLM as the "winner," for generating a greater number executable tests, we want to do an in depth qualitative analysis of the generated code to evaluate its usability. Since our objective for this work is to generate useful OpenMP offloading tests for 3.0 features that are both thorough

Table 5. Number of valid tests according to base language and OpenMP Specification

	Number of valid tests	Number of invalid tests
C	37 (47%)	41 (53%)
Fortran	35 (45%)	43 (55%)



(a) Invalid test categorization for C tests generated.

(b) Invalid test categorization for Fortran tests generated.

Fig. 2. Distribution of Invalid tests per category

and compliant with the OpenMP specification, we analyze each test file to see if the code adheres to the OpenMP specification and the restrictions within.

To avoid being overly critical towards the generated tests we consider a test to be valid even if it checks for *single* aspects of the OpenMP directive or clause correctly. For example, for the the `firstprivate` clause; the requirement is that the variable is privatized and that it is assigned the value of the original list item. Even if one of these aspects is checked by the generated test exercising the `firstprivate` clause, we count it as valid. For the invalid tests we sorted them by the principle reason for their incorrectness. One aspect of which is discussed in Sec. 4.1 where the use of `target` directive conflicts with the semantics of the OpenMP 3.0 feature (include all tests that exercise the `copyin` clause). Although more than one error was found in majority of the invalid offloading tests, we have classified them into four broad categories namely; (1) incorrect use of OpenMP directive, (2) incorrect use of OpenMP clause, (3) OpenMP semantics ignored, or (4) programming error. The distribution for the C and Fortran tests generated is shown in Figures 2a and 2b respectively. Due to space constraints we will discuss the categories along with a few illustrative examples, as observed in the generated tests.

- ***Incorrect use of OpenMP directive.*** Overall, 37% C and 12% Fortran invalid tests fell in this category. In some tests, like those checking the ordering of threads using the `ordered` directive, the result was not dependent on the effect of the directive. In another test exercising `atomic` directive with the `capture` clause the atomic

```
// Offload computation to target device
    #pragma omp target map(tofrom: results)
    {
        #pragma omp parallel for schedule(static, 2)
        for (i = 0; i < n; i++) {
            results[i] = i * 2;
        }
    }
```

Listing 4. Main body of the offlading test for `for` directive with the `schedule` clause generated by ChatGPT

operation read from distinct array indices to a private variable that do not need protection. The code would have worked as expected even without the use of the `atomic` directive.

- ***Incorrect use of OpenMP clause.*** An interesting example in this category is the test generated for the `parallel` directive with `if` clause. Here the generated test did not check if parallelism was achieved based on the conditional evaluated in the `if` clause. Another example is the test generated for the `for` directive with the `schedule` clause. The schedule clause specifies how collapsed iterations of a worksharing-loop construct are divided into chunks and distributed among threads of an OpenMP team of threads. OpenMP allows for five kinds of schedules; `auto`, `dynamic`, `guided`, `runtime`, and `static`. The generated test shown in Listing 4 uses the `static` schedule with a chunk size of two. This means that each OpenMP thread executes two consecutive iterations in a round-robin manner till the iteration space is covered by the executing threads. The test does not check if the static schedule was honored by the executing threads, as it does not capture the information regarding the thread number that executes the iteration.
- ***OpenMP semantics ignored.*** Critical errors in this category include tests in combination with `default` clause and tests with `sections` directive. The `default` clause determines the implicitly determined data-sharing attributes of variables that are referenced in the construct. The *data-sharing-attribute* that the `default` clause can specify is either `firstprivate`, `private`, `shared` or `none`. Though most of these are self explanatory, particular attention must be given when *data-sharing-attribute* `none` is specified as it implies that the *data-sharing attribute* of the variables used inside the construct are not implicitly determined and hence they must be **explicitly** specified. Generated tests that used *data-sharing-attribute* `none` failed to include explicit *data-sharing-attribute* for the variables used. The tests generated for OpenMP `sections` directive, which as been around since OpenMP 3.0, had a recurring pattern of error unrelated to offloading to device. In the generated tests the iterators were shared across different `section` structured blocks within the same `sections` directive. This causes a race condition and the values computed in the `section` are undefined.
- ***Programming error.*** This category includes generated tests that contain base language errors (mainly observed in the Fortran programs) and errors in setting the "pass" flag. These form a very small part of the invalid test spectrum. For example, the tests exercising `ordered` directive the relies on visual inspection of the print statements and not the pass flag (which is always set to *true*). These could lead to larger number of false-positives if not corrected.

4.2 Correcting Invalid Tests

We tried to fix all the above mentioned errors that do not require a complete rewrite of the test code. We identified four tests that could not be fixed without a complete rewrite. Depending on the complexity of the changes required to make the corrections we roughly characterize the effort as *small*, *medium* or *large*, similar to the “t-shirt sizing” approach

sometimes used in work estimation for software development. These categories do not necessarily correspond to number of lines of code (LOC) changed, which we also record, and may be different depending on the level of OpenMP expertise. Here we present a few interesting cases, one each from the levels of effort, from among the C and Fortran tests that we were able to successfully fix to make them meaningful and valid. The exhaustive list of corrected tests is shown in Table 6 with the level of effort to make the corrections.

Listing 5 shows the main offloading kernel of the generated test for `barrier` directive. There are two conceptual errors in this test (1) the `for` construct has the OpenMP *worksharing* property which has an implicit barrier at exit unless the `nowait` clause is applied and (2) the checks are done inside the `target` region, but the value of the *pass_flag* is not propagated back to the host after the offloading kernel completes execution on the device. Since the *pass_flag* is a scalar variable in the C language, the default behavior is as if is `firstprivate` and only persists during the duration of the offloaded kernel. Due to this the generated test runs to completion but always reports *Failed* when executed even when other errors are corrected. When these issues were fixed (as shown in Listing 6) by adding a `nowait` clause to the `for` construct on line 8, and adding the *pass_flag* with a `tofrom` modifier to the `map` directive, the code reliably tests the `barrier` directive. We classified the effort required for correcting this test as *small* and the corresponding LOC changed is 14.

Listing 7 shows the main offloading kernel of the generated test for the combined `parallel for` directive with the `if` clause. If the expression in the `if` clause evaluates to *false*, then the number of threads executing the parallel region is set to one. Hence, even when `num_threads` is set to greater than one, the `if` clause determines if the the parallel region will be executed by multiple threads. The generated test code does not check if, in fact, the value of the condition had any effect and reports success if the array *results* is updated correctly. We correct this invalid test by updating the value stored in the *results* array such that is reflects the number of threads executing the parallel region. The corrected test checks for both values of the `if` clause (true and false) and only reports *Test Passed* if both scenarios work as per the OpenMP specification. We classified the effort required for correcting this test as *medium* and the corresponding LOC changed is 33.

A more complicated test case is presented in Listings 9 and 10. The `sections` directive allows for a number of unrelated structured block sequences delimited using the `section` construct to be executed concurrently by threads in the team. The `cancel` directive can be used to activate cancellation (i.e. abort the execution) of the innermost enclosing parallel region of the type specified by *cancel-directive-name*, which is `sections`, in this test as shown on line 12 of Listing 9. All threads that belong to the team executing the parallel region check for active cancellation only at *cancellation points*, which only occur at OpenMP barriers, a cancel region, or an explicit `cancellation point`. The generated test is missing any *cancellation point* construct, so threads that do not encounter the `omp section` construct on line 8 have no place where they check whether the execution of the `sections` parallel region has ben canceled.

The corrected code in Listing 10 shows the correct use of the `cancel` directive for `sections`. While one thread cancels the sections (in line 14) all other threads in the parallel region check the status of cancellation at the `cancellation point` (line 20) and record the cancellation of the `sections`. Other non-trivial errors in the generated code are (1) shared iterator *i* which causes race and undefined results, and (2) values of *results1* and *results2* are checked for updates or lack thereof on the host which violates OpenMP semantics, as the array *results1* will have unspecified value because it is not assigned in the lexically last `section` enclosed by the `sections` directive. Effort required for this test is classified as *large* due to the complexity of semantics even when only 30 LOC were changed during the corrections.

Table 6. Effort and lines of code (LOC) change for corrected C and Fortran tests

Effort	Corrected tests	LOC change for C	LOC change for Fortran
small	test_atomic_capture	n/a	6
	test_atomic_read	n/a	4
	test_barrier	14	4
	test_cancel_taskgroup	32	21
	test_cancellation_point	26	24
	test_critical_name	n/a	21
	test_declare_reduction_initializer	n/a	4
	test_declare_simd_aligned	n/a	32
	test_declare_simd_linear	n/a	18
	test_declare_simd_simdlen	n/a	17
	test_declare_simd_uniform	n/a	18
	test_master	7	7
	test_parallel_default	4	n/a
	test_parallel_firstprivate	14	n/a
	test_parallel_for_default	2	n/a
	test_parallel_for_nowait	n/a	6
	test_parallel_sections_default	2	n/a
	test_parallel_sections_nowait	13	6
	test_parallel_sections_num_threads	15	8
	test_parallel_sections_reduction	5	10
	test_sections_nowait	2	6
	test_sections_private	n/a	5
	test_sections_reduction	2	n/a
	test_single_private	n/a	8
	test_single_copyprivate	2	n/a
	test_task_default	n/a	23
	test_task_final	n/a	20
	test_task_firstprivate	n/a	19
	test_task_if	n/a	29
	test_task_mergable	n/a	19
	test_task_priority	n/a	19
	test_task_private	12	n/a
	test_task_shared	n/a	21
	test_taskyield	n/a	19
medium	test_for_ordered	28	15
	test_ordered	19	17
	test_parallel_for_if	33	34
	test_parallel_for_num_threads	17	15
	test_parallel_for_ordered	4	10
	test_parallel_for_private	12	12
	test_parallel_for_schedule	14	24
	test_parallel_sections_if	5	5
	test_task_untied	39	23
	test_taskwait	29	19
large	test_cancel_parallel	26	9
	test_cancel_sections	46	31
	test_for_private	18	4
	test_for_schedule	30	29
	test_parallel_if	8	16
	test_task_depend	21	26

```
// Offload computation to target device
#pragma omp target map(tofrom: results)
{
  #pragma omp parallel shared(results)
  {
    // initalize and get thread id and
     num_threads
    #pragma omp for
    for (i = 0; i < n; i++) {
      results[i] = id;
    }
    #pragma omp barrier
     // Check if all elements have been
     updated correctly
        ...
  }
}
// Check value of pass_flag to report test
     Passed or Failed
```

Listing 5. Generated offloading test for `barrier` directive

```
// Offload computation to target device
#pragma omp target map(tofrom: results, pass_flag)
{
  #pragma omp parallel shared(results, pass_flag)
  {
      //initalize and get thread id and
     num_threads
      #pragma omp for schedule(static, 1) nowait
      for (i = 0; i < n; i++) {
        results[i] = id;
       }
       #pragma omp barrier
       // Check if all elements have been updated
     correctly
          ...
  }
}
// Check value of pass_flag to report test Passed
     or Failed
```

Listing 6. Corrected offloading test for `barrier` directive

```
// Offload computation to target device
#pragma omp target map(tofrom: results)
{
  #pragma omp parallel for if(condition)
   for (i = 0; i < n; i++) {
     results[i] = i * 2;
    }
}
// Check results
for (i = 0; i < n; i++) {
       ...
}
```

Listing 7. Generated offloading kernel for `parallel for` combined directive with `if` clause

```
for(condition = 0;condition < 2; condition++){
// Offload computation to target device
  #pragma omp target map(tofrom: results)
  {
    #pragma omp parallel for if(condition)
     num_threads(4)
    for (i = 0; i < n; i++) {
      results[i] = omp_get_num_threads();
    }
  }
  // Check results
  if(!condition){
        ...
  } else {
        ...
  }//end-else
}//end-for
```

Listing 8. Corrected offloading kernel for `parallel for` combined directive with `if` clause

4.3 Using the Tests to Evaluate OpenMP Implementations on Frontier

Of the 78 generated tests we could correct 74 tests (95%). During the correctness checks we found that 3 of the 74 tests use the `copyin` clause (discussed in 4.1.1) and thus cannot be offloaded due to conflicting semantics. They are relevant only in the *host-only* mode and are not included in the results for offloading tests shown in Figure 3. They are included in the *host-only* results presented in Figure 4.

With the 71 correct offloading tests, we evaluate OpenMP implementations in AMD ROCm (version 17.0.0), HPE/Cray CCE (17.0.0) and GNU GCC (13.2.1-dev) on Frontier super-computer (x86 CPU and AMD MI250X GPUs). Figure 3 shows the number of corrected tests that compiled without errors, ran to completion, passed the validation, and failed validation using the AMD, CRAY and GNU compilers with offloading enabled and mandatory. For completeness, we also compile and run the 71 tests above and the three tests relying on `copyin` in *host-only* mode. In this mode offloading kernels are executed on the host and thus, among other things, does not require associated data transfers.

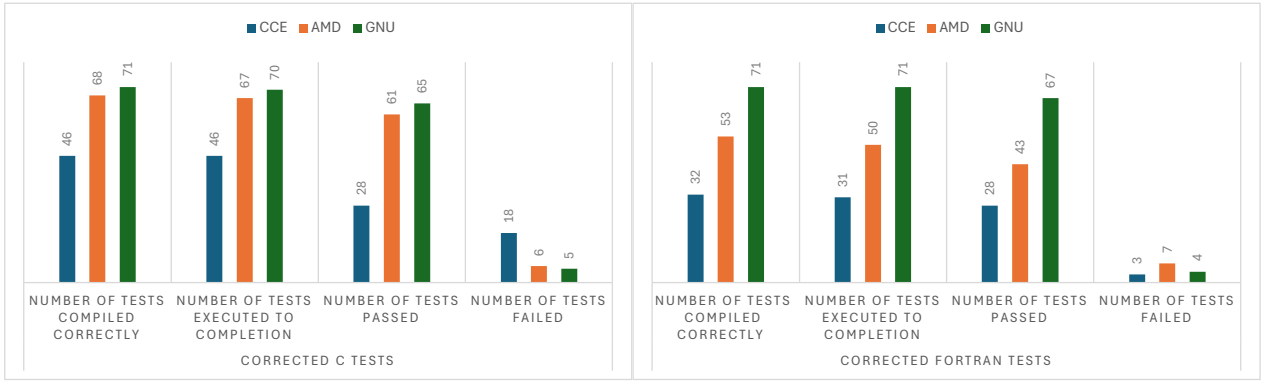
```c
// Set cancel flag to test cancellation
cancel = 1;
// Offload computation to target device
#pragma omp target map(tofrom: results1,
      results2, cancel)
{
  #pragma omp parallel sections
  {
    #pragma omp section
    {
      for (i = 0; i < n; i++) {
        if (cancel && i == 5) {
          #pragma omp cancel sections
        }
        results1[i] = i * 2;
       }
    }

    #pragma omp section
    {
      for (i = 0; i < n; i++) {
        results2[i] = i * 3;
      }
    }
  }
}
// Check value of results1 and results2
     arrays on host
```

Listing 9. Generated offloading kernel for cancellation directive used inside sections

```c
// Set cancel flag to test cancellation
cancel = 1;
// Offload computation to target device
#pragma omp target map(tofrom: results) map(to:
     cancel)
{
  int i;
  #pragma omp parallel private(i)
  {
    #pragma omp sections
    {
      #pragma omp section
      {
        if (cancel == 1) {
          #pragma omp cancel sections
        }
      }
      #pragma omp section
      {
        for (i = 0; i < n; i++) {
      #pragma omp cancellation point sections
          results[i] = i * 3;
        }
      }
    }
  }
}
// Check value of results array on host
```

Listing 10. Corrected offloading kernel for cancellation directive used inside sections

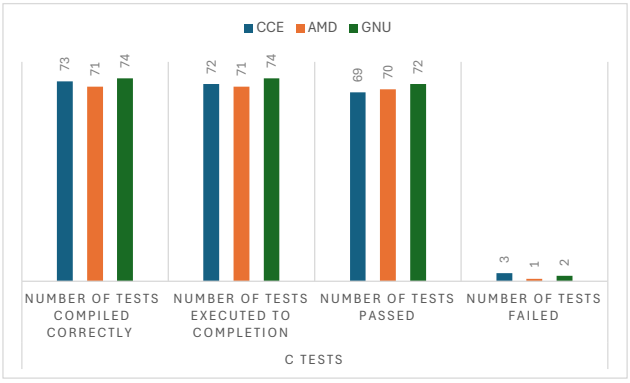


(a) Testing OpenMP implementation for C using the corrected C tests.

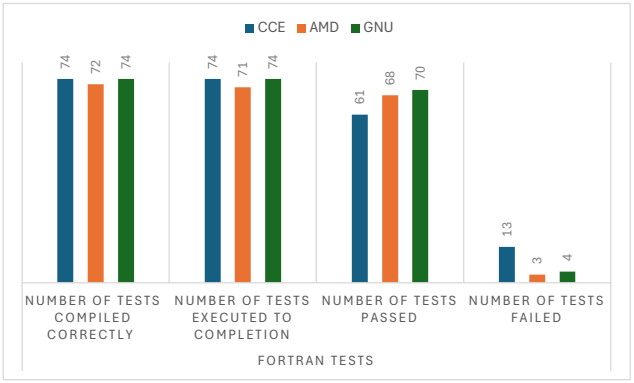
(b) Testing OpenMP implementation for Fortran using the corrected Fortran tests

Fig. 3. Tests compiled and executed with offloading on device by setting OMP_TARGET_OFFLOAD to *mandatory*

Across the board the pass rate is much better for *host* execution. This is particularly noticeable for CCE where 93% of the C and 82% of Fortran feature tests pass the validation on *host* compared to 39% C and Fortran on the *device*. For the Cray CCE compiler, we observe that OpenMP `cancellation, ordered, single`, tasking and reduction tests produce compilation errors when compiled for the device. The AMD OpenMP implementation (based on LLVM) has more



(a) Testing OpenMP implementation for C using the corrected C tests.



(b) Testing OpenMP implementation for Fortran using the corrected Fortran tests

Fig. 4. Tests compiled and executed in host only mode by setting OMP_TARGET_OFFLOAD to *disabled*

uniform results for *host* and *device* execution with a few more failed validation tests for OpenMP cancellation, `ordered` construct, and untied `task`. The best OpenMP conformance is seen in the results from GCC implementation of OpenMP for both *host* (97% C and 95% Fortran) and *device* (92% C and 94% Fortran). These numbers are the current snapshot of the implementation status of different implementations and are only intended to show the gaps in features available and possible problems with incorrect implementations for older OpenMP features in combination with offloading. Since these compilers are under active development, we hope to see more tests pass validation as time passes.

5 RELATED WORK

The use of machine learning to aid in various aspects of code development has received a lot of attention over the last decade. A systematic overview of such use is presented in [11], where the authors survey the landscape of machine learning aided software development. Within the context of HPC, the utility of using LLMs to generate parallel programming kernels is studied in [14], where the authors use GitHub Copilot to generate code for representative HPC kernels (including AXPY, GEMV, GEMM, SpMV, Jacobi Stencil, and CG) using different programming languages and models, and targeting both CPUs and accelerators. The use of LLMs for software testing was studied in [34] where the authors surveyed 102 studies involving various aspects of software testing tasks and the potential and challenge of using various LLM models to aid in testing. Using LLMs to aid in the generation of test suites for compilers was studied in [21], where the authors investigate the use of different LLMs to generate a validation and verification test suite that targets the OpenACC directive. In this work, the authors study the performance of state-of-the-art LLMs, including open-source LLMs - Meta's Codellama, Phind's fine-tuned version of Codellama, Deepseek's Deepseek Coder and closed-source LLMs - OpenAI's GPT-3.5-Turbo and GPT-4-Turbo when tasked to generate a testsuite for compilers implementing the OpenACC offloading directives. Furthermore, the authors investigate fine tuning and various prompt methods, concluding that for this particular task, the Deepseek-Coder-33b-Instruct LLM outperformed other models in the study, followed by GPT-4-Turbo.

An evaluation of ChatGPT-3.5, specifically, for code generation [17] drew test problems primarily from the LeetCode platform.[1] The study used 728 test problems, targeting solution code generation in C, C++, Java, JavaScript, and Python. In testing the effectiveness of using multiple rounds with ChatGPT to fix erroneous programs, they conclude that ChatGPT has problems fixing "wrong answer" errors (as opposed to "wrong details" errors) because "(1) ChatGPT lacks the ability to grasp logical details, and (2) ChatGPT lacks in dealing with problems that require complex reasoning." On the other hand, most (70%) of compilation errors can be successfully addressed through multi-round fixing, suggesting that it might be useful to try multi-round fixing with the OpenMP test cases. They also found that the quality of ChatGPT's generated solutions was very different for test problems published before the cut-off date for the data used to train this version of ChatGPT (2022-01-01) compared to those published after it (acceptance rates of 68% vs. 20%), suggesting the value of having data in the training set directly connected to the problem posed. Note that in this paper, we use ChatGPT-4, and utilize C and Fortran as the base languages for the OpenMP test cases.

Random test generation is a technique with broad applicability. Pankratov [28] illustrates this approach applied to OpenMP, automatically generating programs testing the `omp parallel for` construct. Laguna et al. [15] have also extended their Varity random random test generator to support a small set of OpenMP constructs in the generated programs. Varity's primary focus is on comparative testing for numerical (floating point) discrepancies between different different compilers and/or different hardware platforms.

The Creal tool [16] produces C-language test code through a combination of random generation and functions extracted from real-world code. The authors show that their approach finds compiler bugs which have not been identified by other testing strategies, and characterize it as complementary to other testing approaches. One can imagine this approach being extended to include OpenMP constructs in either the random code or the real-world functions.

6 CONCLUSION

In this work we explored the use of ChatGPT-4 as a building block to generate a suite of tests for OpenMP features. We have chosen a set of directives and clauses, a total of 78 combinations, which first appeared in OpenMP 3.0 (released in May 2008) but are also relevant for accelerators. We prompted ChatGPT to generate tests in the C and Fortran languages for device (accelerator) offloading. We found that of the 78 generated tests 67 C tests and 43 Fortran tests compiled successfully and fewer than those executed to completion. We demonstrate that successful compilation and execution to completion are inadequate metrics for evaluating generated code and that, at this time, even commodity LLMs require expert intervention for code verification. We provide detailed analysis regarding the quality of tests generated. We improved on the usable generated tests (95%) by correcting all the errors. Our experiment successfully built a quality validation suite. In doing so we have analyzed the key error categories that need special attention from users when relying on LLMs for OpenMP code generation. Although general purpose LLMs can generate codes for specialized programming model like OpenMP, manual intervention by experts is necessary to validate the generated codes. This also points to (1) the need for specialized models geared towards HPC tasks which have a better understanding of the semantics of different programming models, and (2) more training data (especially for cases identified in this paper), in the form of correct code and coding practices, to be available for training LLMs. With "small" effort 37% of generated invalid C tests and 63% of generated invalid Fortran tests could be corrected, which helps with programming productivity. Overall, we could use 95% of the generated tests. With the corrected tests, we validate recent implementations from

[1] https://leetcode.com/

HPE, AMD and GNU on the Frontier super-computer to show the missing/incorrect OpenMP feature implementations provided by the different vendors.

ACKNOWLEDGMENTS

This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

REFERENCES

[1] Anthropic. 2023. Claude: An Anthropic AI Assistant. https://www.anthropic.com/index/claude
[2] Thomas Applencourt. 2023. https://github.com/TApplencourt/OvO.
[3] Franck Cappello, Alejandro Duran, Haihang Jin, Sergi Mateo, Christian Terboven, and Stephen L Olivier. 2009. OpenMP 3.0 validation suite. In *Proceedings of the 5th International Workshop on OpenMP*. Springer-Verlag, Berlin, Heidelberg, 153–166.
[4] Sunita Chandrasekaran et al. 2012. A benchmark-based evaluation of OpenMP 3.0 and 4.0 implementations. In *International Workshop on OpenMP*. Springer, Berlin, Heidelberg, 60–72.
[5] Cohere. 2022. Cohere Command R: Retrieval-Augmented Generation. https://cohere.ai
[6] NVIDIA Corporation. 2023. NVIDIA OpenMP Validation Suite. https://developer.nvidia.com/openmp-validation-suite Accessed: 2024-08-15.
[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL] https://arxiv.org/abs/1810.04805
[8] Jose Monsalve Diaz, Swaroop Pophale, Kyle Friedline, Oscar Hernandez, David E Bernholdt, and Sunita Chandrasekaran. 2018. Evaluating Support for OpenMP Offload Features. In *Proceedings of the 47th International Conference on Parallel Processing Companion*. ACM, New York, 31.
[9] Alejandro Duran, Christian Terboven, Dirk Schmidl, Sergi Mateo, and Lars Koesterke. 2009. The Barcelona OpenMP Tasks Suite. In *Proceedings of the 2009 International Conference on Parallel Processing*. IEEE, Washington, DC, 124–131.
[10] EleutherAI. 2021. GPT-Neo: An Open-Source Autoregressive Language Model. https://github.com/EleutherAI/gpt-neo
[11] Enrique, Bappaditya Dey, Sandip Halder, Stefan De Gendt, and Wannes Meert. 2022. Code Generation Using Machine Learning: A Systematic Review. *IEEE Access* 10 (2022), 82434–82455. https://doi.org/10.1109/ACCESS.2022.3196347
[12] GitHub, Inc. 2021. GitHub Copilot. https://github.com/features/copilot Accessed: 2024-08-15.
[13] GNU Project. 2023. GNU OpenMP Validation Test Suite. https://gcc.gnu.org Accessed: 2024-08-15.
[14] William Godoy, Pedro Valero-Lara, Keita Teranishi, Prasanna Balaprakash, and Jeffrey Vetter. 2023. Evaluation of OpenAI Codex for HPC Parallel Programming Models Kernel Generation. In *Proceedings of the 52nd International Conference on Parallel Processing Workshops* (Salt Lake City, UT, USA) *(ICPP Workshops '23)*. Association for Computing Machinery, New York, NY, USA, 136–144. https://doi.org/10.1145/3605731.3605886
[15] Ignacio Laguna, Patrick Chapman, Konstantinos Parasyris, Giorgis Georgakoudis, and Cindy Rubio-González. 2024. Testing the Unknown: A Framework for OpenMP Testing via Random Program Generation. arXiv:2410.09191 [cs.SE] https://arxiv.org/abs/2410.09191
[16] Shaohua Li, Theodoros Theodoridis, and Zhendong Su. 2024. Boosting Compiler Testing by Injecting Real-World Code. *Proc. ACM Program. Lang.* 8, PLDI, Article 156 (June 2024), 23 pages. https://doi.org/10.1145/3656386
[17] Zhijie Liu, Yutian Tang, Xiapu Luo, Yuming Zhou, and Liang Feng Zhang. 2024. No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT. *IEEE Transactions on Software Engineering* 50, 6 (2024), 1548–1584. https://doi.org/10.1109/TSE.2024.3392499
[18] LLVM Project. 2023. LLVM OpenMP Validation Suite. https://llvm.org Accessed: 2024-08-15.
[19] Message Passing Interface Forum. 2021. *MPI: A Message-Passing Interface Standard Version 4.0.* Message Passing Interface Forum. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf
[20] Nikola Mišić and Ivan Dodović. 2024. An Assessment of Large Language Models for OpenMP-Based Code Parallelization: A User Perspective. *Journal of Big Data* 11 (2024), 19. https://journalofbigdata.springeropen.com/articles/10.1186/s40537-024-01019-z Accessed: 2024-12-10.
[21] Christian Munley, Aaron Jarmusch, and Sunita Chandrasekaran. 2024. LLM4VV: Developing LLM-driven testsuite for compiler validation. *Future Generation Computer Systems* 160 (2024), 1–13. https://doi.org/10.1016/j.future.2024.05.034
[22] Oak Ridge National Laboratory and University of Delaware. 2024. OpenMP Validation and Verification Suite. https://github.com/OpenMP-Validation-and-Verification Accessed: 2024-08-15.
[23] OpenAI and Josh Achiam *et al.* 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774
[24] OpenMP Architecture Review Board. 2008. OpenMP Application Program Interface Version 3.0. [Online]. Available: http://www.openmp.org/mp-documents/spec30.pdf.
[25] OpenMP Architecture Review Board. 2013. *OpenMP Application Programming Interface Version 4.0.* https://www.openmp.org/specifications/ Accessed: 2024-08-15.
[26] OpenMP Architecture Review Board. 2018. *OpenMP Application Programming Interface Version 4.5.* https://www.openmp.org/specifications/ Accessed: 2024-08-15.

[27] OpenMP Architecture Review Board. 2021. *OpenMP Application Programming Interface Version 5.2.* https://www.openmp.org/specifications/ Accessed: 2024-08-15.

[28] Pankratov, Svyatoslav. 2018. Automated test generation for optimizing compilers with OpenMP support. *MATEC Web Conf.* 210 (2018), 04014. https://doi.org/10.1051/matecconf/201821004014

[29] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67.

[30] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2023. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. arXiv:1910.10683 [cs.LG] https://arxiv.org/abs/1910.10683

[31] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL] https://arxiv.org/abs/2302.13971

[32] Stanford University. 2023. Stanford Alpaca: An Instruction-following LLaMA Model. https://github.com/tatsu-lab/stanford_alpaca

[33] Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax

[34] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Transactions on Software Engineering* 50, 4 (2024), 911–936. https://doi.org/10.1109/TSE.2024.3368208

[35] BigScience Workshop and Teven Le Scao *et al.* 2023. BLOOM: A 176B-Parameter Open-Access Multilingual Language Model. arXiv:2211.05100 [cs.CL] https://arxiv.org/abs/2211.05100

[36] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V Le. 2019. XLNet: Generalized autoregressive pretraining for language understanding. In *Advances in neural information processing systems*. Curran Associates, Inc., Red Hook New York, 5753–5763.