LLM-Assisted Translation of Legacy FORTRAN Codes to C++: A Cross-Platform Study

Nishath Rajiv Ranasinghe[1], Shawn M. Jones[1], Michal Kucer[1], Ayan Biswas[1], Daniel O'Malley[1], Alexander Buschmann Most[1], Selma Liliane Wanna[1], Ajay Sreekumar[2]
[1]Los Alamos National Laboratory, Los Alamos NM 87545,
[2]School of Information, University of Arizona, 103 E 2nd St 4, Tucson, AZ 85721
Correspondence: ayan@lanl.gov

Abstract

Large Language Models (LLMs) are increasingly being leveraged for generating and translating scientific computer codes by both domain-experts and non-domain experts. Fortran has served as one of the go to programming languages in legacy high-performance computing (HPC) for scientific discoveries. Despite growing adoption, LLM-based code translation of legacy code-bases has not been thoroughly assessed or quantified for its usability. Here, we studied the applicability of LLM-based translation of Fortran to C++ as a step towards building an agentic-workflow using open-weight LLMs on two different computational platforms. We statistically quantified the compilation accuracy of the translated C++ codes, measured the similarity of the LLM translated code to the human translated C++ code, and statistically quantified the output similarity of the Fortran to C++ translation.

1 Introduction

A Large volume of scientific computational software implemented in HPC environments has been written in programming languages such as Fortran and C due to their superior performance. However, recent advancements in computer hardware are not fully utilized by older generations of Fortran, and these legacy codes often encounter difficulties with memory allocations. There is a lack of human resources to maintain and improve these code-bases for mission critical applications in the future (Shipman and Randles, 2023; Pietrini et al., 2024).

Propriety (e.g. ChatGPT) and open weight (e.g. Llama (Touvron et al., 2023)) LLMs have vastly improved code generation (Wang and Chen, 2023) and code translation between modern programming languages (Jiao et al., 2023) due to widespread availability of training examples, but not without difficulties (Pan et al., 2024). As efforts expand to translate scientific software from legacy programming languages to more modern languages via agentic workflows, there is a need for systematic methods to evaluate the effectiveness of machine generated scientific software.

However, very few studies exist for LLM-assisted code translation from Fortran to C++, primarily due to a lack of quality training data sets. A recent study (Lei et al., 2023), compiled pairs of OpenMP Fortran and equivalent C++ codes to evaluate LLM code translation and evaluated their results using both quantitative (e.g., CodeBLEU score (Ren et al., 2020)) and qualitative approaches (e.g., human evaluation). There is also a lack of LLM-based Fortran to C++ code translation tools that can be readily deployed to assist developers in mission critical and secure environments. Furthermore, earlier attempts to translate code from Fortran to C++ have not accounted for successful compiles or output evaluation of the translated code (Theurich et al., 2001).

In this study, we make several contributions. We conduct an analysis of translating open-source code-bases using open-weight models. Our workflow (Figure 1) is designed to be agnostic of any specific LLM or computational platform (e.g., vLLM), building towards a set of standardized evaluation measures for machine-generated code translation. We evaluate the similarity to the human-translated target code using the common CodeBLEU measure (Ren et al., 2020), how much of the translated code compiles (compilation accuracy (Wen et al., 2022a)), and how well the output of the compiled translated code matches the original compiled Fortran code (output similarity). We also categorize any compile errors to demonstrate different behaviors among LLMs. To our knowledge, this is the first attempt to statistically quantify code translation accuracies of open-weight LLMs between computational platforms, the first such study involving Fortran, and the first to apply all of these evaluation techniques together.

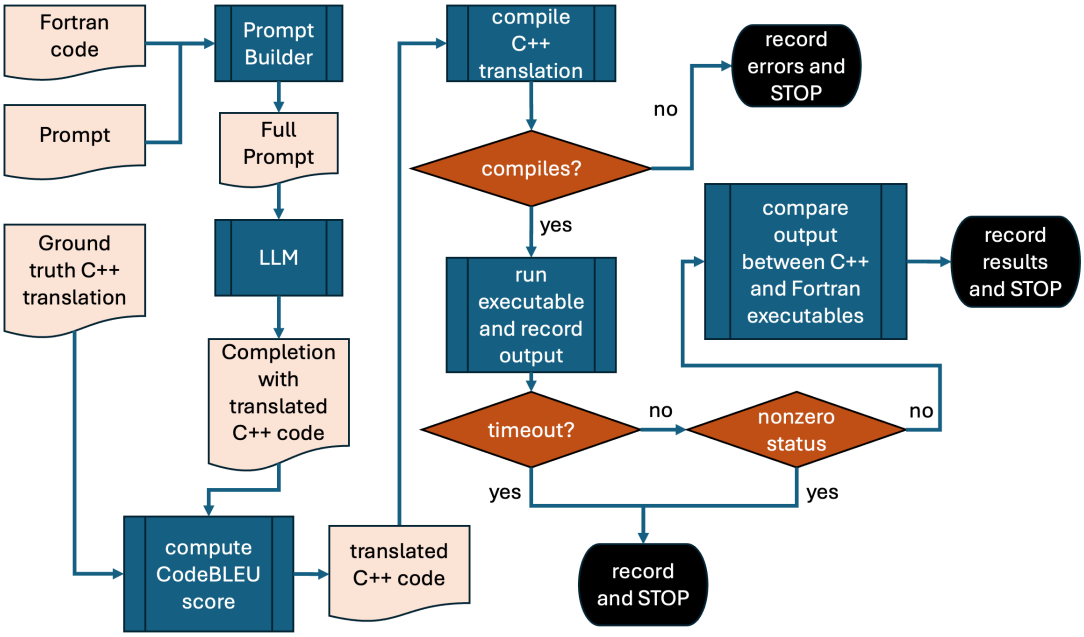


Figure 1: Regardless of LLM, our workflow evaluates several parts of the LLM's code translation, starting by comparing it to a human-translated ground truth with CodeBLEU, then moving to evaluate how well the translation compiles and executes. Finally, the workflow compares the output between the original Fortran code and the translated code's C++ executable.

2 Background

Despite the emergence of numerous modern programming languages, Fortran remains integral in legacy scientific applications, HPC, and areas requiring intensive numerical computations, such as climate modeling (Méndez et al., 2014), computational fluid dynamics (Derlaga et al., 2013), solving inverse problems (Cuer and Bayer, 1980), full waveform inversion (Komatitsch and Tromp, 2002), subsurface flow (Mills et al., 2007), space applications (Ocampo and Senent, 2006), crystallography (Grosse-Kunstleve et al., 2002), radiation transport (Waters et al., 2007) and structural analysis (Nardelli, 1995). Unfortunately, Fortran is no longer a popular language (Shipman and Randles, 2023) and finding assistance from the community for future development is challenging. We chose C++ as a target language because it has more community support, but it also has a number of desirable features for scientific computing in the HPC environment, including its highly efficient feature set, template techniques (Veldhuizen and Jernigan, 1997), the standard template library (Musser and Saini, 1995), and advanced memory management (Attardi et al., 1998). Unfortunately, efforts to translate legacy code-bases from Fortran to C++ have encountered several challenges stemming from differences in language paradigms, syntax, and standard libraries.

LLMs have emerged as an efficient and robust method for translating code between programming languages. Many LLMs exist (de Groot, 2024), and there are different computational platforms (Emani et al., 2023) for executing LLMs. In this work, we evaluate two such platforms: vLLM and SambaNova. vLLM is a library providing a common interface for efficiently serving different LLMs across different hardware architectures utilizing the PagedAttention algorithm (Kwon et al., 2023). SambaNova is an AI accelerator platform that provides specialized hardware for executing LLMs (Prabhakar et al., 2024). We compare both in this paper.

3 Related Work

Fortran to C++ translation has traditionally been conducted manually by experienced programmers. There have been few efforts to convert these legacy code-bases from Fortran to C++ using source-to-source translation tools (Grosse-Kunstleve et al., 2012; Feldman, 1990). However, the translated codes from these sources lack readability and require manual changes to implement memory management functionality (Theurich et al., 2001).

Previous systematic studies of code translation between pairs of modern programming languages

such as C, C++, Go, Java, and Python using LLMs have been met with varying degree of compilation success from 2.1 to 47.3% for code specific (codeGEN, CodeGenX, StarCoder) and text based general purpose (GPT-4, Llama-2, TB-Airboros, TB-Vicuna) LLMs with GPT-4 having the most success (Pan et al., 2024). Recent efforts to create larger code bases of example training data sets for popular and niche programming languages have improved the LLM assisted translations between more modern languages (Yan et al., 2023). A recent study (Chen et al., 2024) utilized an LLM based agentic method that seamlessly integrates multiple verification processes into iterative cycles for translating Fortran to C++. This approach employs a questioner-solver module to delegate referencing and decision-making tasks to separate LLMs, a multi-turn dialogue collection that effectively captures the nuanced aspects of translating and finally fine-tuning of three open-weight LLMs using the data produced to improve the accuracy of the models. Our study differs from theirs (Chen et al., 2024) by evaluating the capabilities of open-weight LLMs that can be readily deployed in a mission critical environment to translate Fortran to C++ on different computational architectures. We also differ by our choice in evaluations. We include compilation accuracy, the translated code's similarity to human translated codes, and a comparison of the similarity of outputs between our ground truth Fortran codes and the translated code from the LLM. Unlike other studies, we also apply the open-source Rosetta code repository (Rosetta Code Community, 2025) as a data source for evaluating the translation of Fortran to C++.

4 Methodology

4.1 Data

To evaluate how well each LLM's translation matches a human translation, we required not only Fortran code, but ground truth C++ translations. We acquired two datasets containing pairs of Fortran and equivalent C++ codes. Rosetta Code (Rosetta Code Community, 2025) provides coding examples for the same programming task in multiple languages. We created a web scraper to produce a dataset of 243 Fortran and their corresponding C++ examples from the Rosetta Code website in October 2023. We retained only those examples for which there was at least one Fortran and corresponding C++ example per programming task. Our second dataset consists of 101 examples from the DataRaceBench (DRB) benchmark (Liao et al., 2017) obtained from the OpenMP Fortran to C++ dataset (Lei et al., 2023) that contains the same code implemented in different languages in support of the benchmark. From each dataset, we selected fully developed 344 computer programs with varying degrees of complexity, to ensure ground truth Fortran and C++ programs compile.

4.2 LLMs

Model parameters in LLMs are preset configurations that determine the model's architecture and training process, such as the number of layers, learning rate, and batch size. The number of parameters varies between LLMs. However, prior work (Hoffmann et al., 2022) demonstrated that the performance of LLMs does not necessarily linearly increase with the number of parameters.

We chose LLMs that are well regarded by industry, can be deployed in a mission-critical environment, allow for local deployment to satisfy privacy concerns, have a diversity of model parameter sizes for comparison, and are also supported by the vLLM and SambaNova Cloud platforms (SambaNova). Table 1 shows the LLMs we selected based on this criteria.

4.3 Workflow

Figure 1 shows the evaluation process we applied to each Fortran code and LLM. We start by building each full prompt by combining each Fortran code with the prompt in Figure 2. Using this full prompt, we requested that each LLM convert the Fortran code to C++. Because LLMs are known to vary their responses due to their stochastic nature, we issued the same prompt multiple times for each Fortran code. We set up vLLM (Kwon et al., 2023) using onsite hardware at the Los Alamos National Laboratory (DGX hardware equipped with 8 A100s NVIDIA GPUs along with 2 AMD EPYC 7742 64-Core Processors) and issued the same prompt 128 times per Fortran code per LLM. We utilized temperature of 0.8, min-p of 0.05, top-p of 0.95, and set the maximum generation length to 8192 tokens across the LLM models. We also used the OpenAI Python API library to prompt Llama models hosted by SambaNova Cloud, which is equipped with SambaNova SN40 Reconfigurable Dataflow Units (RDUs) (Prabhakar et al., 2024). Due to rate limits on the SambaNova Cloud, we only executed the same prompt 25 times per Fortran code

Table 1: The LLMs used in this study.

LLM	# parameters	Computational platform
Open code interpreter	33B	vLLM
Llama 3.1	70B	vLLM
Mistral Large Instruct 2407	123B	vLLM
Llama 3.3	70B	vLLM
Llama 3.1	8B	SambaNova Cloud
Llama 3.1	70B	SambaNova Cloud
Llama 3.1	405B	SambaNova Cloud
Llama 3.3	70B	SambaNova Cloud

```
You are an exceptionally intelligent coding assistant specializing in code
translation, particularly from Fortran to C++. You consistently deliver accurate
and reliable translations while maintaining the original code's functionality and
structure.

Please translate this Fortran code to C++. Follow these guidelines:
1. Maintain the overall structure and functionality of the original code.
2. Use modern C++ practices and idioms where appropriate.
3. Ensure that all functions, subroutines, and modules are properly translated to
their C++ equivalents.
4. Pay attention to differences in array handling, I/O operations, and memory
management between Fortran and C++.
5. Include any necessary C++ libraries or headers.
6. Add comments to explain any significant changes or non-trivial translations.

Please return the translated C++ code in one code block.
Please restrict your output to the translated code only.
```

Figure 2: The prompt used in this study.

per LLM. We utilized temperature of 0.8, top-p of 0.9, and context length of 4096 across the Llama models in the SambaNova Cloud. From each completion, we recorded the C++ code and compared it to the ground truth C++ code from our datasets via CodeBLEU score (Ren et al., 2020). From there, we evaluated the Fortran code's compilation accuracy and output similarity.

4.4 Similarity to human translated code

CodeBLEU (Ren et al., 2020) measures how well a machine translation matches a human translation for the same code. The CodeBLEU score contains four dimensions of comparison: matching n-grams, matching weighted n-grams, Abstract Syntax Tree matching, and data-flow analysis. We apply the human ground truth translation from each dataset to arrive at a CodeBLEU score. We perform bias analysis of the translated C++ codes across various LLMs, as an indicator of the code translation quality. We use CodeBLEU scores of the human translated C++ codes with their corresponding machine translated codes. In our scenario, since we run the same translation command prompt for a given code multiple times and we might get variations in the code translation, our bias analysis takes into account this stochasticity in LLM-based code generation. To perform this, for each LLM, we first calculate individual average CodeBLEU scores for each ground truth Fortran file across the trials. Since CodeBLEU depicts similarity, we calculate bias (that represents error) as $Bias = 1 - CodeBLEU$. With this formulation, now we can use these averaged bias scores to approximate a distribution using a non-parametric Kernel Density Estimate (KDE) approach(Chen, 2017). In this method, there exist different choices for its kernel types; such as Gaussian, triangular, rectangular, and the Epanechnikov kernel (Gramacki, 2018). Generally, variations due to kernel types are considered to be less significant compared to the choice of kernel bandwidth (Silverman, 2018). Silverman's rule of thumb for bandwidth selection generally produces smooth and good-quality density estimation (Biswas et al., 2016). We use this approach in our work and generate the KDE plots, as shown in Figure 3a for vLLM based translated codes and Figure 3b shows the KDE plots for the SambaNova Cloud based translated codes.

Table 2: Classification of compiler errors used in this work.

Compile Error Category	Error topic	String matches from g++ compiler
Syntax Error	Missing operators, missing delimiters, incorrect usage of tokens, or anything else resulting from poor programming syntax	expected before error: no match for ‘operator>= stray “’ in program error: void value not ignored as it ought to be error: ‘std::std’ has not been declaredcannot be used as a function error: assignment of read-only locationerror: invalid initialization of non-const reference of type error: lvalue required as increment operand error: no matching function for call to error: missing terminating " character error: too many arguments to function
Type Error	An issue with use of data types	invalid conversion cannot convert
Linker Error	The implied use of external libraries	is not a member of ‘std’ error: aggregate ‘std::stringstream ss’ has incomplete type and cannot be defined undefined reference
Declaration Error	Declaring variables before use	error: too many initializers was not declared has not been declared
Semantic Error	Proper application of functions or operators	invalid operands invalid use of
Scope Error	Using variable outside of their established scope	not in this scope is not captured
Template Error	Invalid use of C++ templates	wrong number of template arguments
File and I/O Error	the code refers to nonexistent filesystem resources	No such file or directory
Memory Error	Incorrect use of memory operations	invalid use of delete
Other Error	Anything else not covered with the string matching above	

4.5 Success of compilation

Compilation accuracy of the translated C++ measures how many translations successfully compile without errors (Wen et al., 2022b). We compiled each translated C++ using the g++ v5.3.0 compiler on Red Hat Enterprise Linux Workstation release 7.9. If a C++ translation failed to compile, we recorded the compiler output and did not proceed further with that translation (Figure 1). We reviewed the compiler output and categorized each error as shown in Table 2. The

4.6 Similarity of outputs

Output similarity compares the output of each Fortran program to that of its C++ translation generated from the LLM. We compiled each Fortran program and ran the resulting executable to capture its output. Then, we did the same with each LLM-generated C++ translation that successfully compiled. Outputs from scientific programs consist of text and numeric data. Humans may look at two outputs and consider them the same where a direct string match would score them radically different (e.g., `b(50,50)= 0.00000000` vs. `b(50,50)= 0.0` and `Fib for 30 832040` vs. `Fib for 30 = 832040.0`). We first tokenized each output using the NLTK (Bird et al., 2009) `word_tokenize` function to produce a list of strings. Then, we attempted to convert each token to a floating point number using the Python `float` function. If the token could be converted, we rounded it to a precision of 4 decimal places. If not, then we left the token as a string. We, then applied a Jaro-Winkler (Jaro, 1989; Winkler, 1990) score to each set of tokens to measure their similarity.

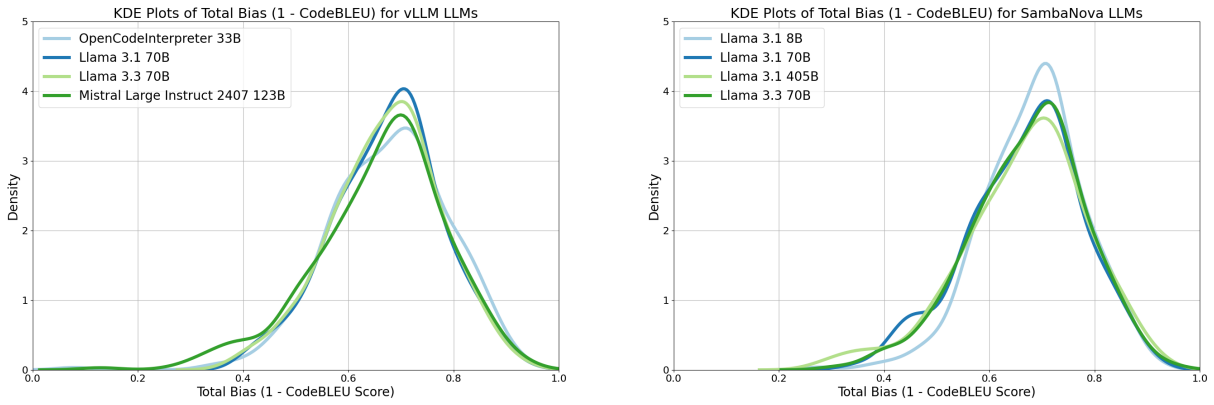
Thus, by the end of the workflow we have evaluated each translation in comparison to a human translation, how well it compiles, and whether it produces the same output as the Fortran submitted to the system at the beginning.

5 Results and Discussion

5.1 Similarity to human translated code

CodeBLEU scores demonstrate how well an LLM's code translation matches a human translation of the same code. Figure 3 shows the bias of CodeBLEU scores between LLMs. Scores on the x-axis provide a distance between LLM generated C++ translations and their human ground truth equivalents. Higher scores that indicate that the translation is farther than the ground truth and thus a poorer match. At first glance Figure 3 appears to show that there is not much difference between LLMs, but the peaks give a more nuanced story.

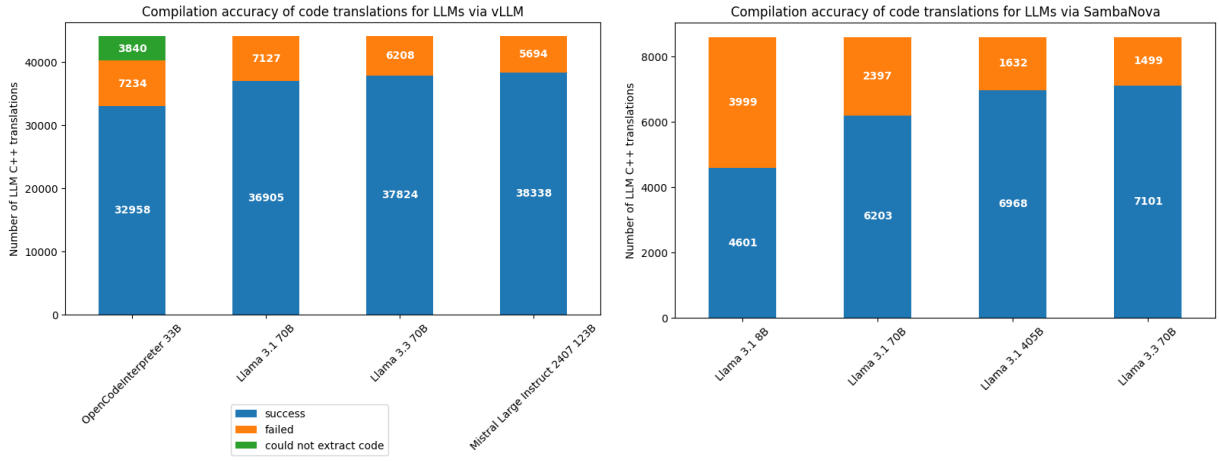
Figure 3a shows that Llama 3.1 70B leads with the highest rate of translations that do not match human ground truth. OpenCodeInterpreter 33B



(a) vLLM

(b) SambaNova

Figure 3: Kernel density estimate plots demonstrating the distribution of total bias (1 - CodeBLEU Score) for each Fortran translation demonstrates different distributions per execution platform.



(a) vLLM

(b) SambaNova

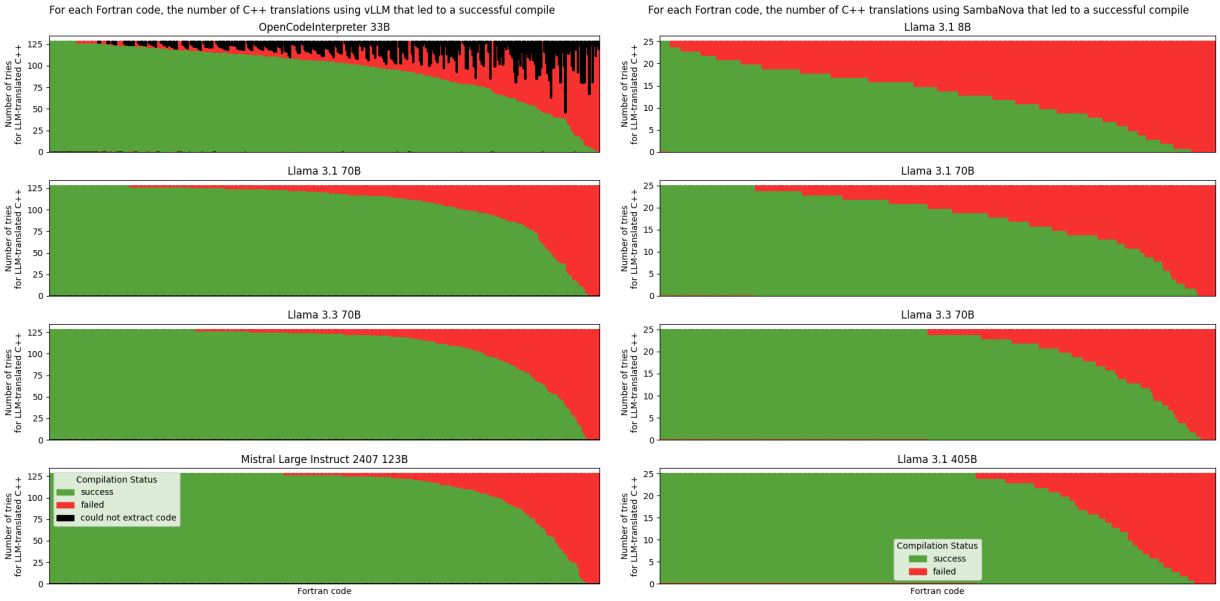
Figure 4: Compilation accuracy of each LLM by execution platform shows that the increase in the number of model parameters is proportional to the increase in compilation accuracy.

(Zheng et al., 2025) has the lowest peak outperforming Mistral Large (AI, 2025). Mistral does have a small peak lower on the x-axis, indicating many more that might be closer to human ground truth.

SambaNova has a similar peak in Figure 3b, indicating a higher number of LLM translations that do not match human ground truth. Llama 3.1 8B's CodeBLEU bias is highest. Thus, its translations are least consistent with human translations. In contrast, Llama 3.1 405B has the lowest peak, but appears only marginally better in consistency than other models. These results with the commonly-used CodeBLEU metric demonstrate that larger models provide translations closer to human ground truth, but the amount of similarity in these distributions necessitate our other measures to more clearly separate performance.

5.2 Success of compilation

Figure 4 shows the compilation accuracy results for each computational platform and LLM. In both cases, we see an increase in the number of successful compiles as one increases the number of parameters in the LLM. Additionally, as seen in Figure 4a, while the LLMs served by vLLM appear to generate more successfully compilable code, OpenCodeInterpreter generates completions from which we cannot extract code. In contrast, SambaNova's results in Figure 4b show no instances where LLM completions produced code that could not be extracted. Additionally, we see that, for vLLM, Llama 3.1 70B and Llama 3.3 70B have comparable performance. This is not the case with these two LLMs on SambaNova Cloud, where Llama 3.1 405B and Llama 3.3 70B have similar performance.



(a) vLLM (b) SambaNova

Figure 5: Each Fortran code is plotted along the x-axis while the count of tries for a corresponding C++ translation is placed on the y-axis. Translations that compiled successfully are shown in green, and those that failed are marked in red. Note same Fortran code is not always shown at the same point in the x-axis. Compilation accuracy of each translated Fortran program differs per model with some LLMs having more difficulty translating certain codes than others. We note that LLMs with a higher number of parameters have more success per Fortran code.

Figure 5 demonstrates the distribution of compilation accuracy for all Fortran codes. These sandcharts represent each Fortran code on the x-axis. The y-axis represents each translation of that code into C++. Green shows translations that successfully compile. Red shows failures. By executing each LLM multiple times we can see the level of variation in their responses and note that not all translation failures occurred equally. Some translations were always successfully compiled while others were more varied. We also note the same pattern of improving compilation accuracy among all Fortran codes as the number of parameters increases across models. vLLM shows more consistent translations (green rising closer to the top) while SambaNova shows a dramatic improvement for Llama 3.1 405B over Llama 3.3 70B that was not apparent in the raw numbers shown in Figure 5b.

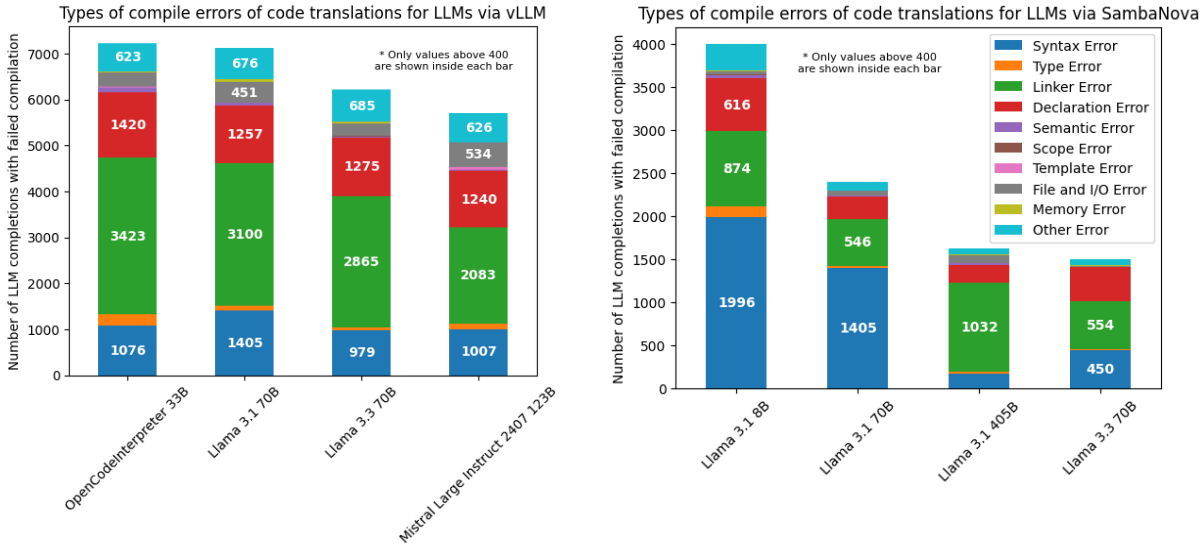
Figure 6 shows the distribution and categorization of of compile failures. In Figure 6a, most of the compile errors generated from the LLMs served in vLLM are linker errors, representing the assumed inclusion of libraries not specified via an `#include` directive. In contrast, in Figure 6b the majority of the compile errors shown for LLMs served in SambaNova Cloud are syntax errors. Again, we see that Llama 3.3 70B and Llama 3.1 405B have comparable performance, though their compile error distribution varies.

5.3 Similarity of outputs

Figure 7 shows the distribution of Jaro-Winkler scores comparing the outputs of the ground truth Fortran programs to the outputs of their LLM C++ translations. We note the same familiar pattern of increasing number of parameters leads to better mean similarity of inputs. Mistral Large with vLLM in Figure 7a and Llama 3.1 405B with SambaNova in Figure 7b both outperform Llama 3.3 70B in this case. Mistral Large, however produces a tighter distribution of similar outputs.

6 Conclusion

We conducted an analysis of how well open-weight LLMs translate open-source code-bases from Fortran to C++. We presented an LLM-independent and platform-independent workflow for our evaluation. This workflow evaluates several elements of translation quality. We consider the similarity between human ground truth and machine translation, if the translated C++ code compiles, what errors are encountered if the compile fails, and finally how well the resulting C++ translation's ex-



(a) vLLM

(b) SambaNova

Figure 6: Distribution of compile error categories for each C++ translation shows that LLMs produce different errors in their translated code.

ecutable produces the same output as the original Fortran code.

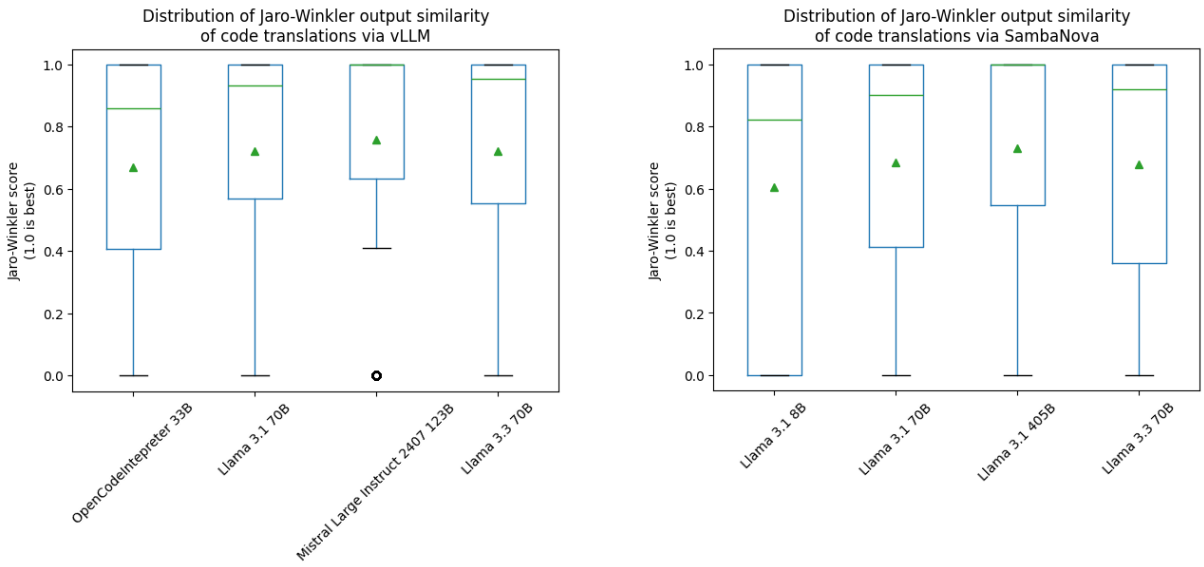
We ran this workflow with LLMs on both the vLLM and SambaNova Cloud platforms. Because LLMs do not always produce the same output each time, we ran 128 instances of the same translation on vLLM and 25 on SambaNova to ensure we had a sizeable sample space. Unsurprisingly, we discovered that those LLMs with higher model parameter counts tend to produce better results. Our codeBLEU analysis reveals that Mistral Large served on vLLM and Llama 3.1 405B served on SambaNova Cloud produce codes that better matches human translations. Our compilation evaluation demonstrates that Mistral Large on vLLM and Llama 3.1 405B on SamaNova Cloud have higher counts of compilable code, with Llama-3.3 70B being comparable. We demonstrated that not all Fortran codes were translated consistently, showing that some LLMs produced C++ translations that more consistently compiled for a given Fortran code. We also found that the translated codes from vLLM that failed to compile mostly had linker errors while those from SambaNova largely contained syntax errors, even for the same LLM model. Finally, we showed that, for successful compiles, the output of the translated executables better matched the output of the original Fortran with Mistral Large on vLLM and Llama 3.1 405B on SambaNova Cloud, with Llama 3.3 70B being comparable on both platforms.

The implications for scientific computing are mixed. The state of the art shows the code bases in Fortran can be translated to C++ readily, but also demonstrate that no LLM on either platform was free of error. We still require a human-in-the-loop for code translation.

7 Limitations

While our study presents a workflow for systematic evaluation of open-weight LLMs for Fortran-to-C++ code translation, there are several limitations that must be acknowledged: Our evaluation workflow is not yet packaged into a standalone tool that can provide Fortran-to-C++ translations along with compilation statistics and output similarity. Automating this workflow would make scientific discovery more accessible for researchers working in HPC environments. We did not present our attempts to improve compilation accuracy through agentic workflows by incorporating the error messages generated from compiling the codes produced by the LLM into a automatic dialog with the LLM. Our initial efforts in that direction were shown to increase the compilation accuracies of the translated codes and we are pursuing the agentic workflows in a future study.

Additionally, our study could be enhanced by incorporating more complex and extensive Fortran code-bases, such John Burkardt's data set (Burkardt, Accessed: 2025-01-30) which are highly



(a) vLLM

(b) SambaNova

Figure 7: Distribution of Jaro-Winkler scores for output similarity comparison between original Fortran executables and LLM C++ executables. Green triangles represent means while green lines are medians.

relevant to scientific computing. Furthermore, Chen et al. (Chen et al., 2024) showed that fine-tuning LLMs on Fortran to C++ datasets could improve each model's CodeBLEU scores by 1.5 to 3.3 times with up to a 92% increase in successful compilations. Focusing our study's analysis on models which have been fine-tuned for Fortran to C++ translation could help create more useful tools for developers.

Further improvements could be made with prompt design and in this study, we used the same prompt for every LLM. It is possible that further exploration of prompt design could uncover that different models perform better with different prompts (Liu et al., 2023; Knobloch et al., 2025). Our study focused solely on open-weight LLMs such as Llama and Mistral. While comparisons do exist for both natural language translation as well as coding (without translating), our literature review found a lack of studies comparing open-weight LLMs to proprietary models like GPT and Gemini for code translation. Including these models, along with the source-to-source translation tools (Feldman, 1990; Grosse-Kunstleve et al., 2012) which were popular for Fortran to C++ in the past could provide a clearer benchmark for our results. Additionally, in this study, we did not test the capabilities of the new generation of reasoning models (OpenAI's o1, o1-mini, o3-mini; DeepSeek-R1; and Anthropic Claude 3.7 Sonnet) to translate Fortran to C++. However, our workflow delivers a plug-and-play solution to test any LLMs code translation capabilities on any computational platform without any modifications.

In this study, we did not consider improving code translation accuracy using few-shot learning via Retrieval Augmented Generation (RAG) as it is studied elsewhere (Bhattarai et al., 2024).

8 Acknowledgments

This work was supported by the Computational Systems and Software Environments subprogram of National Nuclear Security Administration's (NNSA's) Advanced Simulation and Computing program through Los Alamos National Laboratory (LANL). LANL is operated by Triad National Security, LLC, for the National Nuclear Security Administration of U.S. Department of Energy (Contract No. 89233218CNA000001). This research used resources provided by the Darwin testbed and DGX pod at LANL which is funded by the Computational Systems and Software Environments subprogram of LANL's Advanced Simulation and Computing program (NNSA/DOE). We are also grateful to SambaNova Systems, Inc for providing access to SambaNova Cloud and technical support. This work is approved for unlimited release with an LA-UR number LA-UR-25-21128.

References

Mistral AI. 2025. Mistral large 2407. Accessed: 2025-01-30.

Giuseppe Attardi, Tito Flagella, and Pietro Iglio. 1998. A customisable memory management framework for c++. *Software: Practice and Experience*, 28(11):1143–1184.

Manish Bhattarai, Javier E. Santos, Shawn Jones, Ayan Biswas, Boian Alexandrov, and Daniel O'Malley. 2024. Enhancing code translation in language models with few-shot learning via retrieval-augmented generation. *Preprint*, arXiv:2407.19619.

Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.".

Ayan Biswas, Guang Lin, Xiaotong Liu, and Han-Wei Shen. 2016. Visualization of time-varying weather ensembles across multiple resolutions. *IEEE transactions on visualization and computer graphics*, 23(1):841–850.

John Burkardt. Accessed: 2025-01-30. John burkardt's homepage. `https://people.sc.fsu.edu/~jburkardt/`. Accessed: 2025-01-30.

Le Chen, Bin Lei, Dunzhi Zhou, Pei-Hung Lin, Chunhua Liao, Caiwen Ding, and Ali Jannesari. 2024. Fortran2cpp: Automating fortran-to-c++ migration using llms via multi-turn dialogue and dual-agent integration. *Preprint*, arXiv:2412.19770.

Yen-Chi Chen. 2017. A tutorial on kernel density estimation and recent advances. *Biostatistics & Epidemiology*, 1(1):161–187.

Michel Cuer and Roger Bayer. 1980. Fortran routines for linear inverse problems. *Geophysics*, 45(11):1706–1719.

Marie de Groot. 2024. Are those 700,000 Large Language Models (LLMs) on Hugging Face really necessary?

Joseph M Derlaga, Tyrone Phillips, and Christopher J Roy. 2013. Sensei computational fluid dynamics code: a case study in modern fortran software development. In *21st AIAA Computational Fluid Dynamics Conference*, page 2450.

Murali Emani, Sam Foreman, Varuni Sastry, Zhen Xie, Siddhisanket Raskar, William Arnold, Rajeev Thakur, Venkatram Vishwanath, and Michael E Papka. 2023. A comprehensive performance study of large language models on novel ai accelerators. *arXiv preprint arXiv:2310.04607*.

Stuart I Feldman. 1990. A fortran to c converter. In *ACM SIGPLAN Fortran Forum*, volume 9, pages 21–22. ACM New York, NY, USA.

Artur Gramacki. 2018. *Nonparametric kernel density estimation and its computational aspects*, volume 37. Springer.

Ralf W Grosse-Kunstleve, Nicholas K Sauter, Nigel W Moriarty, and Paul D Adams. 2002. The computational crystallography toolbox: crystallographic algorithms in a reusable software framework. *Applied Crystallography*, 35(1):126–136.

Ralf W Grosse-Kunstleve, Thomas C Terwilliger, Nicholas K Sauter, and Paul D Adams. 2012. Automatic fortran to c++ conversion with fable. *Source code for biology and medicine*, 7:1–11.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. 2022. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*.

Matthew A Jaro. 1989. Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. *Journal of the American Statistical Association*, 84(406):414–420.

Wenxiang Jiao, Wenxuan Wang, Jen-Tse Huang, Xing Wang, and Zhaopeng Tu. 2023. Is chatgpt a good translator? yes with gpt-4 as the engine.

Benjamin Knobloch, Christine Sweeney, Ayan Biswas, and Shawn M. Jones. 2025. Metadata tracking and analysis of llm-based source-to-source code translation. In *Proceedings of the 2025 Conference on Data Analysis*.

Dimitri Komatitsch and Jeroen Tromp. 2002. Spectral-element simulations of global seismic wave propagation—i. validation. *Geophysical Journal International*, 149(2):390–412.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Bin Lei, Caiwen Ding, Le Chen, Pei-Hung Lin, and Chunhua Liao. 2023. Creating a dataset for high-performance computing code translation using llms: A bridge between openmp fortran and c++. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE.

Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. 2017. Dataracebench: a benchmark suite for systematic evaluation of data race detection tools. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA. Association for Computing Machinery.

Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM computing surveys*, 55(9):1–35.

Richard Tran Mills, Chuan Lu, Peter C Lichtner, and Glenn E Hammond. 2007. Simulating subsurface flow and transport on ultrascale computers using pflotran. In *Journal of physics: conference series*, volume 78, page 012051. IOP Publishing.

David R Musser and Atul Saini. 1995. *The STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison Wesley Longman Publishing Co., Inc.

Mariano Méndez, Fernando G. Tinetti, and Jeffrey L. Overbey. 2014. Climate models: Challenges for fortran development tools. In *2014 Second International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*, pages 6–12.

M Nardelli. 1995. Parst95–an update to parst: a system of fortran routines for calculating molecular structure parameters from the results of crystal structure analyses. *Applied Crystallography*, 28(5):659–659.

Cesar Ocampo and Juan Senent. 2006. The design and development of copernicus: A comprehensive trajectory design and optimization system. In *57th International Astronautical Congress*, pages C1–4.

Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA. Association for Computing Machinery.

Rocco Pietrini, Marina Paolanti, and Emanuele Frontoni. 2024. Bridging eras: Transforming fortran legacies into python with the power of large language models. In *2024 IEEE 3rd International Conference on Computing and Machine Intelligence (ICMI)*, pages 1–5. IEEE.

Raghu Prabhakar, Ram Sivaramakrishnan, Darshan Gandhi, Yun Du, Mingran Wang, Xiangyu Song, Kejie Zhang, Tianren Gao, Angela Wang, Xiaoyan Li, Yongning Sheng, Joshua Brot, Denis Sokolov, Apurv Vivek, Calvin Leung, Arjun Sabnis, Jiayu Bai, Tuowen Zhao, Mark Gottscho, David Jackson, Mark Luttrell, Manish K. Shah, Zhengyu Chen, Kaizhao Liang, Swayambhoo Jain, Urmish Thakker, Dawei Huang, Sumti Jairath, Kevin J. Brown, and Kunle Olukotun. 2024. Sambanova sn40l: Scaling the ai memory wall with dataflow and composition of experts. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, page 1353–1366. IEEE.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Rosetta Code Community. 2025. Rosetta code: Programming chrestomathy site. Accessed: 2025-03-03.

SambaNova. Sambanova cloud. Accessed: March 8, 2025.

Galen M Shipman and Timothy C Randles. 2023. An evaluation of risks associated with relying on fortran for mission critical codes for the next 15 years. Technical report, Los Alamos National Laboratory (LANL), Los Alamos, NM (United States).

Bernard W Silverman. 2018. *Density estimation for statistics and data analysis*. Routledge.

G. Theurich, B. Anson, N.A. Hill, and A. Hill. 2001. Making the fortran-to-c transition: how painful is it really? *Computing in Science Engineering*, 3(1):21–27.

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. *Preprint*, arXiv:2302.13971.

Todd L Veldhuizen and M Ed Jernigan. 1997. Will c++ be faster than fortran? In *International Conference on Computing in Object-Oriented Parallel Environments*, pages 49–56. Springer.

Jianxun Wang and Yixiang Chen. 2023. A review on code generation with llms: Application and evaluation. In *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*, pages 284–289.

Laurie S Waters, Gregg W McKinney, Joe W Durkee, Michael L Fensin, John S Hendricks, Michael R James, Russell C Johns, and Denise B Pelowitz. 2007. The mcnpx monte carlo radiation transport code. In *AIP conference Proceedings*, volume 896, pages 81–90. American Institute of Physics.

Yuanbo Wen, Qi Guo, Qiang Fu, Xiaqing Li, Jianxing Xu, Yanlin Tang, Yongwei Zhao, Xin Hu, Zidong Du, Ling Li, Chao Wang, Xuehai Zhou, and Yunji Chen. 2022a. Babeltower: Learning to auto-parallelized program translation. In *International Conference on Machine Learning*.

Yuanbo Wen, Qi Guo, Qiang Fu, Xiaqing Li, Jianxing Xu, Yanlin Tang, Yongwei Zhao, Xing Hu, Zidong Du, Ling Li, Chao Wang, Xuehai Zhou, and Yunji Chen. 2022b. BabelTower: Learning to auto-parallelized program translation. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 23685–23700. PMLR.

William E Winkler. 1990. String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage.

Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. 2023. Codetransocean: A comprehensive multilingual benchmark for code translation. *arXiv preprint arXiv:2310.04951*.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2025. Opencodeinterpreter: Integrating code generation with execution and refinement. *Preprint*, arXiv:2402.14658.