Comparing Llama-2 and GPT-3 LLMs for HPC kernels generation

Pedro Valero-Lara[1,*][0000−0002−1479−4310],
Alexis Huante[2][0009−0008−2818−0265]
Mustafa Al Lail[2][0009−0000−0326−6363],
William F. Godoy[1][0000−0002−2590−5178],
Keita Teranishi[1][0000−0001−6647−2690],
Prasanna Balaprakash[1][0000−0002−0292−5715],
Jeffrey S. Vetter[1][0000−0002−2449−6720]

[1] Oak Ridge National Laboratory, Oak Ridge, TN, 37830, USA
[2] Texas A&M International University, Laredo, Texas 78041, USA
[*]Corresponding author: `valerolarap@ornl.gov`

Abstract. We evaluate the use of the open-source Llama-2 model for generating well-known, high-performance computing kernels (e.g., AXPY, GEMV, GEMM) on different parallel programming models and languages (e.g., C++: OpenMP, OpenMP Offload, OpenACC, CUDA, HIP; Fortran: OpenMP, OpenMP Offload, OpenACC; Python: numpy, Numba, pyCUDA, cuPy; and Julia: Threads, CUDA.jl, AMDGPU.jl). We built upon our previous work that is based on the OpenAI Codex, which is a descendant of GPT-3, to generate similar kernels with simple prompts via GitHub Copilot. Our goal is to compare the accuracy of Llama-2 and our original GPT-3 baseline by using a similar metric. Llama-2 has a simplified model that shows competitive or even superior accuracy. We also report on the differences between these foundational large language models as generative AI continues to redefine human-computer interactions. Overall, Copilot generates codes that are more reliable but less optimized, whereas codes generated by Llama-2 are less reliable but more optimized when correct.

Keywords: LLM · HPC · Llama-2 · GPT.

1 Introduction

Generative-AI large language models (LLMs) are transforming the software industry by automating manual tasks, such as developing, testing, and deploying applications. The use of LLMs could lead to faster and more cost-effective software development. LLMs are also revolutionizing entertainment, education, and healthcare industries by creating realistic images, text, music, and code. However, there are social and ethical concerns surrounding LLMs, including the risk of deep fakes being created and distributed as misinformation or to harm individuals. Therefore, the risks and benefits of LLMs must be carefully considered before widespread adoption.

The emergence of exascale computing presents a challenge in developing software for high-performance computing (HPC) systems owing to the varying hardware and programming models in these complex architectures. To address this challenge, AI-assisted code generation could be used. LLMs can generate code in commonly used programming languages, including C++, Fortran, Python, and Julia. This innovation could make software development for HPC more efficient and manageable. However, limitations exist for AI-assisted code generation given it may only sometimes produce code that is as efficient or reliable as human-written code. The current state of practice, the limitations, and the potential of LLMs must be fully understood to realize their benefits.

The effort described in this paper builds on our previous work [7], in which we investigated the effectiveness of OpenAI Codex for generating HPC code for various numerical kernels in different programming languages and models, including C++, Fortran, Python, and Julia. The study found that the output of OpenAI Codex for C++ is closely linked to the popularity and sophistication of programming models. For example, OpenMP [17] and CUDA [15] received high scores because they are widely used and well-established programming models. However, HIP [1] received lower scores because it is a newer programming model that is not as widely used. The study also found that prompts in Fortran or Python can benefit from incorporating code keywords. However, Julia's prompts perform adequately without the need for code keywords for its mature HPC programming models.

This paper also describes our evaluation of Meta AI's LLM (Llama-2) for generating HPC kernels. The version of Llama-2 we used has 70 billion parameters and was provided by Hugging Chat, an open-source chat bot platform that relies on LLMs to power its conversations. This platform is built on top of the Hugging Face ecosystem. Our evaluation involves generating code for three fundamental numerical kernels: AXPY, GEMV, and GEMM. We then test the resulting 144 kernel codes in four programming languages, C++, Fortran, Python, and Julia, by using various programming models and compilers. These included OpenMP, OpenACC [16], CUDA, HIP, numpy [21], Numba [12], cuPy [14], pyCUDA [10], Julia's Base Threads [11], CUDA.jl [3], and AMDGPU.jl [18].

The paper is organized as follows: Section 2 provides an overview of related efforts that have brought attention to these topics in computer science. Section 3 outlines our methodology for generating and evaluating the code with Llama-2. In Section 4, we present the results of our evaluation and our findings for each language, kernel, and programming model along with additional keyword inputs on the generated outputs. Finally, Section 5 presents our conclusions.

2 Related Work

The Generative Pre-trained Transformer 3 (GPT-3) [5] is a game changer in the evolution of human-computer interactions. Developed by OpenAI,[3] GPT-3 is the third generation of the prediction-based foundational LLM used for

[3] https://openai.com/

several AI-generated, human-like text applications. GPT-3 is used in several natural language processing tasks [9], including ChatGPT, due in part to the large investment ($12 million USD) and size of its training model (175 billion parameters at 800 GB). Hence, GPT-3 and its successor GPT-4[4] are defining several societal questions for the near future. Today, we are at the beginning of a race to develop the best LLM model. In addition to GPT, we can find recently released foundational LLMs such as Llama-2 [20] and PaLM 2[5].

As we enter the exascale computing era, which is dominated by the extreme heterogeneity of hardware and programming models [23], AI-assisted code generation could play a key role in how we develop, deploy, and test software that targets HPC systems. Traditional human-readable code in languages such as C++ [19], Fortran [2], Python [22], and more recently Julia [4], are a straightforward application for LLM's capabilities—capabilities that could help redefine software development. In fact, this rapidly evolving field was recently surveyed in our previous work [7], in which we evaluated the performance of the GPT-3 descendant OpenAI Codex for HPC kernel generation by using GitHub Copilot for several parallel programming models. The quality of the responses depends largely on the number of repositories and programming model maturity. Nichols et al. [13] fine-tuned the use of LLMs to improve the generation of OpenMP pragmas in parallel algorithm implementations, including MPI cases. Chen et al. [6] presented LM4HPC, a framework to conduct HPC-specific tasks in the context of LLMs, and highlighted the lack of training and evaluation datasets in HPC. Hence, we expect to see more work in the convergence of HPC and generative AI via LLMs because of the field's rapid evolution. To the best of our knowledge, this is the first evaluation of Llama-2 for the generation and correctness of HPC kernels and comparison to our baseline from previous work.

3 Methodology

First, we use prompts similar to those in our previous research [7], which are simple prompts based on the programming language, kernel, and programming model. The quality of the prompt is important because it determines how the LLM will generate the requested code based on the information provided. So, several adjustments were made to the prompt until Llama-2 was outputting the code requested. Importantly, the output from the LLM also depends on the data used to train the model. For example, the LLM may not be trained well enough for a particular language or model and may therefore produce inaccurate code no matter the prompt given.

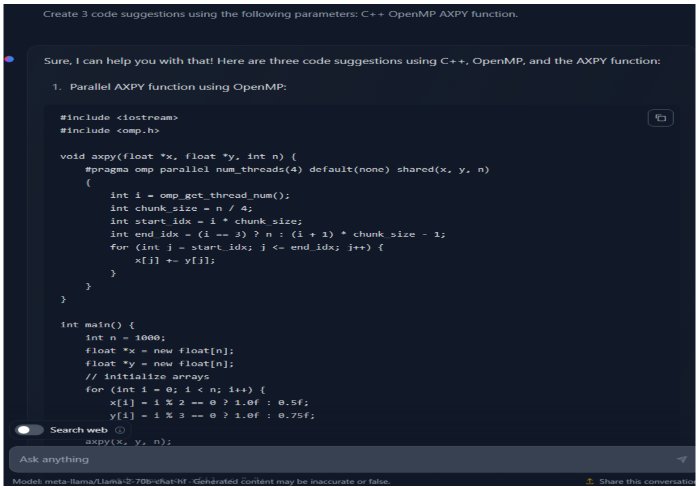
The methodology used in this study involves two main characteristics that will be discussed in the next subsections: (1) how we prompted Llama-2 for code generation based on the kernel, parallel programming model, and programming language and (2) a code correctness metric that will be evaluated by expert observation.

[4] `https://openai.com/product/gpt-4`

[5] `https://ai.google/discover/palm2/`

3.1 Experimental Setup

For our experiments, we used the Hugging Chat website, which, as of August 2023, uses the largest model of Llama-2 called Llama-2-70B. We created an account on Hugging Face to access the necessary features. As shown in Figure 1, the website features a chat box for the user to input their query for the LLM.



Create 3 code suggestions using the following parameters: C++ OpenMP AXPY function.

Sure, I can help you with that! Here are three code suggestions using C++, OpenMP, and the AXPY function:

1. Parallel AXPY function using OpenMP:

```
#include <iostream>
#include <omp.h>

void axpy(float *x, float *y, int n) {
    #pragma omp parallel num_threads(4) default(none) shared(x, y, n)
    {
        int i = omp_get_thread_num();
        int chunk_size = n / 4;
        int start_idx = i * chunk_size;
        int end_idx = (i == 3) ? n : (i + 1) * chunk_size - 1;
        for (int j = start_idx; j <= end_idx; j++) {
            x[j] += y[j];
        }
    }
}

int main() {
    int n = 1000;
    float *x = new float[n];
    float *y = new float[n];
    // initialize arrays
    for (int i = 0; i < n; i++) {
        x[i] = i % 2 == 0 ? 1.0f : 0.5f;
        y[i] = i % 3 == 0 ? 1.0f : 0.75f;
    axpy(x, y, n);
```

Search web
Ask anything
Model: meta-llama/Llama-2-70b-chat-hf · Generated content may be inaccurate or false.
Share this conversatio

Fig. 1. Hugging Chat website interface.

An example of the prompt and the generated code on Llama-2 is illustrated in Figure 1. The structure of the prompt is as follows:

- Create 3 code suggestions using the following parameters: ⟨Programming Language⟩ ⟨Programming Model⟩ ⟨Kernel⟩ ⟨Keyword⟩.
- Create 3 code suggestions using the following parameters: ⟨Programming Language⟩ ⟨Programming Model⟩ ⟨Kernel⟩.

Unlike our previous study based on the GitHub Copilot model [7], which can provide one or more codes, we must specify the number of code suggestions we want when using Llama-2. Importantly, the first prompt is used for C++, Fortran, and Python, whereas the second prompt is used only for Julia. This is because, according to previous research, they determined there was slight sensitivity in Julia prompts when adding a keyword [7]. For the ⟨Kernel⟩ section, instead of prompting "GEMV" or "GEMM," we used the full form of the abbreviations, which are "general matrix-vector multiply" and "general matrix-matrix multiply," respectively. This is because Llama-2 does not interpret what the abbreviations mean. Additionally, Llama-2 has a character limit, so when prompting for three code suggestions, sometimes it could not finish all three codes. Whenever this was the case, we prompted the LLM to continue with the code generation by saying, "please continue with the code," "you stopped, please continue," or similar.

Next, Table 1 lists all the programming languages, programming models, and keywords used in this study. We used the AXPY, GEMV, and GEMM kernels for every programming model. These kernels correspond to one specific operation of the three different levels of the Basic Linear Algebra Subprograms (BLAS) library:[6] the AXPY level-1 BLAS routine computes a scalar-vector multiplication, the GEMV level-2 BLAS routine computes a matrix-vector multiplication, and the GEMM level-3 BLAS routine computes a matrix-matrix operation. The BLAS library operations increase in complexity with each level. Also, the higher the level of the BLAS routine, the more possibilities for optimizations.

We used a total of 48 prompts, which resulted in 144 codes generated by Llama-2. These codes will be evaluated by the correctness metric described in the next subsection, and we will compare the results to those of the LLM Copilot model from earlier work [7].

Kernels: AXPY, GEMV, GEMM		
Programming Language	Programming Model	Keyword
C++	OpenMP	function
	OpenMP(offload)	function
	OpenACC	function
	CUDA	function
	HIP	function
Fortran	OpenMP	subroutine
	OpenMP(offload)	subroutine
	OpenACC	subroutine
Python	numpy	def
	Numba	def
	pyCUDA	def
	cuPy	def
Julia	Threads	
	CUDA	
	AMDGPU	

Table 1. Parameters used for code generation

3.2 Correctness metric

To evaluate the correctness of the generated codes, we use the simple metric approach from our previous work [7]. We consider five levels of correctness and proficiency labels between [0], or *non-knowledge*, and [1], or *expert*, when observing the suggested answers provided by Llama-2 for each combination in Table 1.

0 *non-knowledge*: No code at all or not a single correct code.

[6] `https://www.netlib.org/blas/`

0.25 *novice*: One correct code, but it includes other several correct or incorrect programming models (e.g., OpenACC suggestions in an OpenMP prompt).
0.5 *learner*: One correct code, and there are other incorrect codes, but all of them use the requested programming model.
0.75 *proficient*: All codes are correct and use the programming model requested.
1 *expert*: Only one piece of code is provided, and it is totally correct.

As mentioned, to make the analysis similar to our previous study on the GitHub Copilot LLM, and to obtain more than one code from Llama-2, we must specify the number of codes that we want. So, we will use the highest metric (expert) for cases in which Llama-2 generates all the three requested codes and does so correctly.

4 Results

The following subsections describe our evaluation of the HPC kernels generated by the Llama-2 LLM for four different programming languages: C++, Fortran, Julia, and Python. The code generated by Llama-2 has also been collected and uploaded to a GitHub repository.[7]

4.1 C++

C++ has become the primary programming language used for heterogeneous HPC architectures due to the support that the open-source and vendor communities provide in terms of programming models and compilers. Examples include OpenMP, OpenACC, and CUDA, among others such as HIP, Kokkos, and SYCL. In this study, we focused on the most popular, mature, and widely used programming models in the HPC community: OpenMP, OpenACC, CUDA, and HIP.

OpenMP OpenMP is considered the de facto standard for parallel programming. The OpenMP codes generated by Llama-2 have the highest quality among the C++ codes. Notably, Llama-2 can leverage relatively advanced OpenMP techniques, including tasking (`#pragma omp task`), atomic operations (`#pragma omp atomic update`), and single instruction multiple data (SIMD) primitives (`#pragma omp simd`), among others (`#pragma omp critical`). However, not all codes are correct. Also, in some cases, the OpenMP code provided used a defined number of threads. This is very dependent on the architecture to be used. In general, the number of threads should be equal to the number of cores (`#pragma omp parallel num_threads(4)`). In some particular cases, in the codes corresponding to the AXPY kernel, Llama-2 provided codes that, although similar to the operation conducted by this BLAS routine, were not exactly the same. For instance, the codes did not use a scalar, or they computed other operations,

[7] `https://github.com/mustafalail/Llama-2-70b-HPC-Kernels`

such as dot product. This is not the same for the other operations evaluated (i.e., matrix-vector and matrix-matrix multiplication) in which the codes provided were correct and functional.

We also see significant errors for the OpenMP target offloading case. In most cases, the code generated was a mix of CUDA and OpenMP codes. Also, the OpenMP primitives used did not correspond to OpenMP target offloading. Unlike the previous case, all generated codes were incorrect.

OpenACC A similar scenario is observed in the OpenACC case for the AXPY operation. All codes provided were incorrect and were a mix of CUDA codes with OpenACC primitives. However, much higher quality was found in the other two kernels, in which the OpenACC primitives were effectively used. Indeed, we see some advanced techniques, such as the use of "collapse" to enroll two nested and independent for loops (`#pragma acc loop independent collapse(2)`). Also, we see an effective movement of data between CPU and GPU in some codes and an effective use of tiling/blocking to decompose the matrices. In this case, the codes provided for the kernels of the matrix-vector and matrix-matrix multiplications were correct.

HIP For HIP codes, we found the same error in most of the codes that correspond to the computation of the thread index (`int ind = hipBlockDim_x * hipBlockIdx_x + hipThreadIdx_x;`). In some cases, this index was not even computed or it was only partially computed. This relatively simple error breaks the entire code, even if the rest of the code is correct. Other common errors found include using the same names for both CPU and GPU memory pointers, using bi-dimensional blocks of threads to launch the kernels when the kernel implementation only uses uni-dimensional blocks of threads (or vice-versa), and the wrong use of GPU shared memory. Also, as in the previous OpenMP and OpenACC analyses, we saw a mix of HIP and CUDA codes. In this case, we found errors in all three test cases (AXPY, matrix-vector, and matrix-matrix multiplications).

CUDA Although we found better quality codes for CUDA than for HIP, the Llama-2-generated CUDA codes still contained some important errors. For instance, using `__device__` function decorators for the kernels implementation when the correct decorator is `__global__`, wrong name of CUDA library functions (`hipCublasSdot`), and initializing GPU memory arrays from the CPU are just a few examples of the errors found. However, all of these errors were found in the AXPY kernel. The code generated for the other two kernels was correct and free of errors. In fact, we observed the effective use of important optimization techniques, such as shared memory (`__shared__ float smem[32][32];`) and registers (`register float rA[32];`), which are used to implement relatively complex algorithms based on tiling/blocking for matrix computation.

4.2 Fortran

Fortran was one of the first widely used programming languages for HPC back in the 1970s. In fact, with reasonably good support for current HPC standards, Fortan is still an important programming language for HPC. In the Fortran community, there are two predominant parallel programming models: (1) OpenMP, which is more focused on providing parallel codes for CPUs, and (2) OpenACC, which is more focused on GPUs.

OpenMP Unlike the C++ codes generated by Llama-2 for OpenMP, we see much better results from Llama-2 when generating Fortran code for OpenMP, especially for the AXPY routine. All generated codes were correct and made use of a scalar. Also, the code generated for the other two kernels used the OpenMP decorators efficiently. Notably, although no advanced OpenMP primitives (e.g., SIMD, collapse) were used, relatively highly optimized algorithms based on tiling/blocking for the matrix-matrix multiplication kernels were used. Unfortunately, this was not the case for OpenMP target offloading, a case in which all the codes provided did not make correct use of the OpenMP primitives.

OpenACC For OpenACC, Llama-2 provided the wrong OpenACC codes for AXPY kernels and used OpenMP decorators instead of OpenACC ones. Better codes were generated for the other two kernels, and at least one functional code was provided. The OpenACC primitives were not used correctly in many cases, and some of the primitives used do not even exist in the OpenACC standard.

4.3 Julia

Julia provides a dynamic, compiled front end to LLVM to target scientific computing and data science. Julia's use in HPC is still an area of active exploration [8] and community engagement. In this section, we evaluate the correctness of three different Julia packages: Base.Threads.jl, CUDA.jl, and AMDGPU.jl, which are used for parallel programming on CPUs, NVIDIA GPUs, and AMD GPUs, respectively.

Base.Threads.jl For the parallel CPU codes that use the Base.Threads.jl Julia package, we found that at least one code provided correct matrix-vector and matrix-matrix multiplication. Unfortunately, this is not the case for the AXPY kernel, and all the codes provided for AXPY were invalid. This could be because of Julia's novelty as a programming language in HPC. Notably, in some cases, it can be challenging to generate different codes that implement exactly the same requested operation, such as AXPY using Base.Threads.jl. Common errors found here include missing keywords (`@threads`) or the use of other packages (`Distributed.jl`).

CUDA.jl and AMDGPU.jl The codes generated using the CUDA package (CUDA.jl) were incorrect. Notably, the generated codes attempted to decorate the nested loops that correspond to the kernels in a way that is similar to how they are decorated when using the Base.Threads.jl package. However, using CUDA.jl is not much different from classic CUDA (i.e., the kernels must be implemented out of the main code, and these must be called/launched by using a very specific syntax [`CUDA.@sync @cuda threads = threads blocks = blocks kernel(x...)`]). We found exactly the same issues for the Llama-2-generated AMDGPU.jl codes.

4.4 Python

Python is a high-level, interpreted, general-purpose programming language. The Python community is one of the biggest software communities today together with C and C++.[8] In this study, we used the most popular parallel solutions in the Python ecosystem: numpy, cuPy, pyCUDA, and Numba. Like with C++, the codes generated by Llama-2 for the AXPY kernels were incorrect, and they did not compute the AXPY operation. And again, unlike the AXPY case, Llama-2 provided much better codes for matrix-vector and matrix-matrix multiplication kernels when using numpy and Numba in particular.

Notably, the quality of these successful cases lies in the use of optimization techniques, such as the decomposition of the matrices into chunks or doing stridden memory accesses. However, we found an error that is common in all of the codes generated for cuPy: using the `__shared__` decorator for the GPU functions instead of the `__device__` decorator, which is the one that must be used. Unfortunately, although the rest of the code is correct, this relatively small error breaks the entire code.

4.5 Llama-2 versus Copilot

This section compares the results of the GitHub Copilot model against the results presented above. For the Copilot model, we use the results presented by W. Godoy et al. [7]. The codes generated by the Copilot model are hosted in a GitHub repository.[9]

First, we focus on C++. Figure 2 illustrates the results (correctness) of the C++ codes generated for OpenMP, OpenMP offload, OpenACC, CUDA, and HIP. As shown, Copilot can provide at least one correct code for most of the kernels and programming models, whereas Llama-2 provided correct codes for OpenMP, OpenACC, and CUDA. Although Llama-2 was unable to provide correct codes for OpenMP offload and HIP, the codes that it did correctly generate were higher quality (i.e., optimized) than the ones generated by Copilot.

For Fortran (Figure 3), we have a similar conclusion to that of the C++ study, with the exception of the AXPY kernel. Here, we actually see that Llama-2 achieved much better performance for the AXPY kernel. Once again, however,

[8] `https://www.tiobe.com/tiobe-index/`

[9] `https://github.com/keitaTN/Copilot-hpc-kernels`

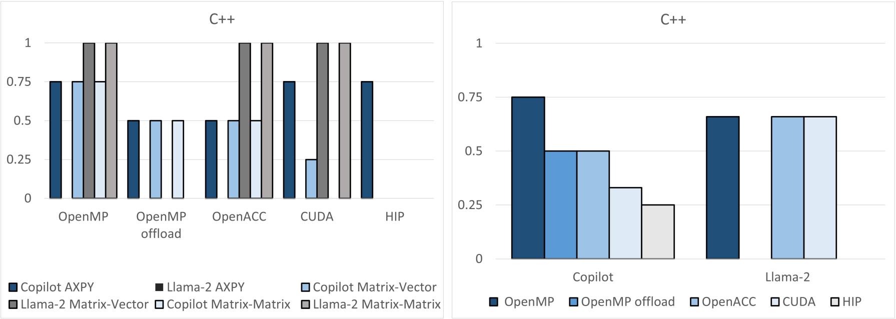


Fig. 2. Results for C++ kernels (left) and programming models (right).

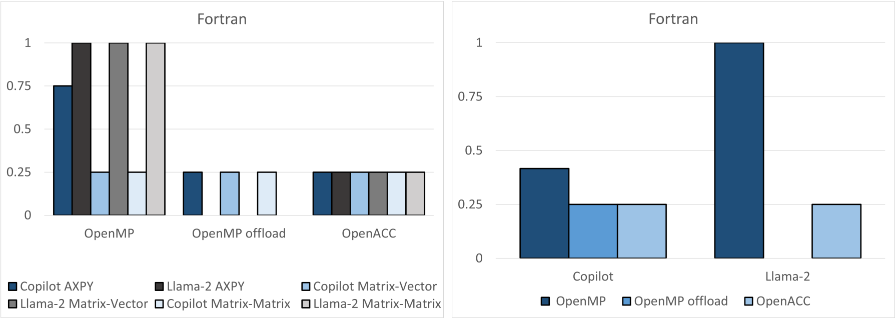


Fig. 3. Results for Fortran kernels (left) and programming models (right).

Llama-2 provided very poor performance for OpenMP offload. Notably, Llama-2 generated high-quality OpenMP codes for all kernels. Copilot still generated at least one correct code for all kernels and programming models and provided the same quality except for the AXPY-OpenMP test case.

For Julia, Llama-2 did not generate correct codes for any of the test cases with the exception of the matrix-vector and matrix-matrix multiplications using the Base.Threads.jl Julia package. This case contained at least one correct code (Figure 4). Unlike Llama-2, GitHub Copilot provided correct codes for all tests except for AMDGPU.jl, for which neither LLM was able to generate correct codes.

Finally, Figure 5 illustrates the results for the Python codes. Copilot was able to generate at least one correct code for most of the test cases with the exception of the level-2 and level-3 BLAS kernels using Numba. Llama-2 provided the best results for these cases. Llama-2 also generated correct results for some numpy codes.

Overall, the main difference between the Copilot and Llama-2 LLMs is that, although Copilot can provide at least one correct code for most of the programming languages and models (albeit the generated codes are not optimized), Llama-2 is more aggressive in terms of optimizations, thereby providing well-

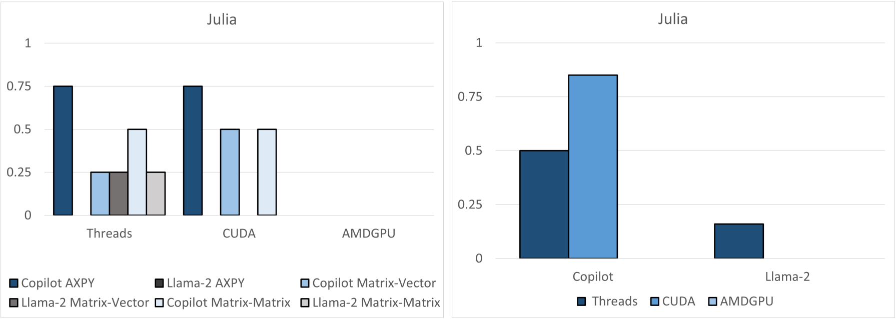


Fig. 4. Results for Julia kernels (left) and programming models (right).

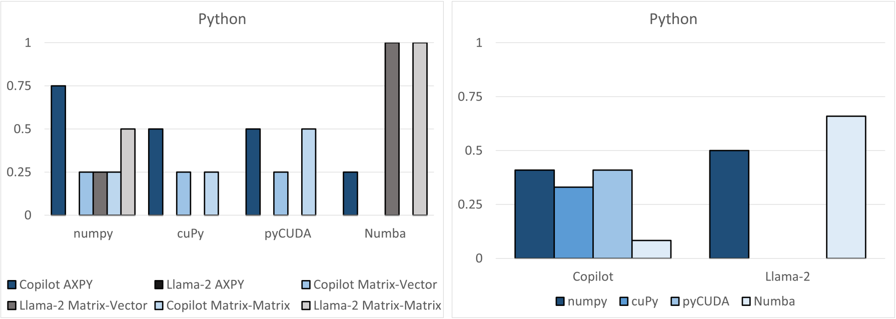


Fig. 5. Results for Python kernels (left) and programming models (right).

optimized codes at the cost of generating incorrect codes in multiple cases. So, in general, Copilot generates codes that are more reliable but less optimized, and codes generated by Llama-2 are less reliable but more optimized.

5 Conclusions

We evaluated the Llama-2 model as an HPC code generator for different programming languages (e.g., C++, Fortran, Julia, and Python) and models used for multicore CPUs (e.g., OpenMP, Base.Threads.jl), NVIDIA GPUs (e.g., CUDA, CUDA.jl, OpenACC, numpy, cuPy, pyCUDA, and Numba), and AMD GPUs (e.g., HIP and AMDGPU.jl).

Llama-2 can provide good-quality HPC codes for some of the previously mentioned solutions. When compared with GitHub Copilot, we realized that the Llama-2 model attempts to provide more optimized codes at the cost of not being as reliable as Copilot. In this study, Llama-2 was able to generate at least one correct code for 40% (C++), 66% (Fortran), 22% (Julia), and 33% (Python) of the test cases. GitHub Copilot provided at least one correct code in 80% (C++), 100% (Fortran), 66% (Julia), and 83% (Python) of the same test cases.

Acknowledgment

This work is funded by Bluestone, an X-Stack project in the DOE Advanced Scientific Computing Office with program manager Hal Finkel.

References

1. AMD: AMD ROCm v5.2 Release (June 2022), `https://rocmdocs.amd.com/en/latest/Current_Release_Notes/Current-Release-Notes.html#amd-rocm-v5-2-release`
2. Backus, J.W., Heising, W.P.: Fortran. IEEE Transactions on Electronic Computers **EC-13**(4), 382–385 (1964). https://doi.org/10.1109/PGEC.1964.263818
3. Besard, T., Foket, C., De Sutter, B.: Effective extensible programming: Unleashing Julia on GPUs. IEEE Transactions on Parallel and Distributed Systems (2018). https://doi.org/10.1109/TPDS.2018.2872064
4. Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B.: Julia: A fresh approach to numerical computing. SIAM Review **59**(1), 65–98 (Jan 2017). https://doi.org/10.1137/141000671
5. Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., Amodei, D.: Language models are few-shot learners. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., Lin, H. (eds.) Advances in Neural Information Processing Systems. vol. 33, pp. 1877–1901. Curran Associates, Inc. (2020), `https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf`
6. Chen, L., Lin, P.H., Vanderbruggen, T., Liao, C., Emani, M., de Supinski, B.: LM4HPC: Towards Effective Language Model Application in High-Performance Computing (2023)
7. Godoy, W., Valero-Lara, P., Teranishi, K., Balaprakash, P., Vetter, J.: Evaluation of OpenAI Codex for HPC Parallel Programming Models Kernel Generation. In: Proceedings of the 52nd International Conference on Parallel Processing Workshops. p. 136–144. ICPPW '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3605731.3605886, `https://doi.org/10.1145/3605731.3605886`
8. Godoy, W.F., Valero-Lara, P., Dettling, T.E., Trefftz, C., Jorquera, I., Sheehy, T., Miller, R.G., Tallada, M.G., Vetter, J.S., Churavy, V.: Evaluating performance and portability of high-level programming models: Julia, Python/Numba, and Kokkos on exascale nodes. In: IEEE International Parallel and Distributed Processing Symposium, IPDPS 2023 - Workshops, St. Petersburg, FL, USA, May 15-19, 2023. pp. 373–382. IEEE (2023). https://doi.org/10.1109/IPDPSW59300.2023.00068, `https://doi.org/10.1109/IPDPSW59300.2023.00068`
9. Hirschberg, J., Manning, C.D.: Advances in natural language processing. Science **349**(6245), 261–266 (2015). https://doi.org/10.1126/science.aaa8685, `https://www.science.org/doi/abs/10.1126/science.aaa8685`
10. Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., Fasih, A.: Pycuda and pyopencl: A scripting-based approach to gpu runtime code generation. Parallel Computing **38**(3), 157–174 (2012). https://doi.org/https://doi.org/10.1016/j.parco.2011.09.001

11. Knopp, T.: Experimental multi-threading support for the julia programming language. In: 2014 First Workshop for High Performance Technical Computing in Dynamic Languages. pp. 1–5. IEEE (2014)
12. Lam, S.K., Pitrou, A., Seibert, S.: Numba: A LLVM-based Python JIT compiler. In: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC. pp. 1–6 (2015)
13. Nichols, D., Marathe, A., Menon, H., Gamblin, T., Bhatele, A.: Modeling parallel programs using large language models (2023)
14. Nishino, R., Loomis, S.H.C.: Cupy: A numpy-compatible library for nvidia gpu calculations. 31st confernce on neural information processing systems **151**(7) (2017)
15. NVIDIA: CUDA Toolkit Documentation - v11.7.0 (May 2022), `https://developer.nvidia.com/cuda-toolkit`
16. OpenACC Architecture Review Board: OpenACC application program interface version 3.1 (November 2020), `https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.1-final.pdf`
17. OpenMP Architecture Review Board: OpenMP application program interface version 5.2 (November 2021), `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf`
18. Samaroo, J., Churavy, V., Phillips, W., Ramadhan, A., Barmparesos, J., Tag-Bot, J., Räss, L., Schanen, M., Besard, T., Smirnov, A., Arakaki, T., Antholzer, S., Alessandro, Elrod, C., Raayai, M., Hu, T.: JuliaGPU/AMDGPU.jl: v0.4.1 (Aug 2022). https://doi.org/10.5281/zenodo.6949520, `https://doi.org/10.5281/zenodo.6949520`
19. Stroustrup, B.: The C++ programming language. Pearson Education (2013)
20. Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Canton-Ferrer, C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P.S., Lachaux, M., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E.M., Subramanian, R., Tan, X.E., Tang, B., Taylor, R., Williams, A., Kuan, J.X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., Scialom, T.: Llama 2: Open foundation and fine-tuned chat models. CoRR **abs/2307.09288** (2023). https://doi.org/10.48550/arXiv.2307.09288, `https://doi.org/10.48550/arXiv.2307.09288`
21. Van Der Walt, S., Colbert, S.C., Varoquaux, G.: The numpy array: a structure for efficient numerical computation. Computing in science & engineering **13**(2), 22–30 (2011)
22. Van Rossum, G., et al.: Python programming language. In: USENIX annual technical conference. vol. 41, pp. 1–36. Santa Clara, CA (2007)
23. Vetter, J.S., Brightwell, R., Gokhale, M., McCormick, P., Ross, R., Shalf, J., Antypas, K., Donofrio, D., Humble, T., Schuman, C., Essen, B.V., Yoo, S., Aiken, A., Bernholdt, D., Byna, S., Cameron, K., Cappello, F., Chapman, B., Chien, A., Hall, M., Hartman-Baker, R., Lan, Z., Lang, M., Leidel, J., Li, S., Lucas, R., Mellor-Crummey, J., Jr., P.P., Peterka, T., Strout, M., Wilke, J.: Extreme heterogeneity 2018 - productive computational science in the era of extreme heterogeneity: Report for DOE ASCR workshop on extreme heterogeneity. Tech. rep., USDOE Office of Science (SC) (United States) (2018). https://doi.org/10.2172/1473756