Evaluation of OpenAI Codex for HPC parallel programming models kernel generation

William F. Godoy, Pedro Valero-Lara, Keita Teranishi, Prasanna Balaprakash, and Jeffrey S. Vetter
Oak Ridge National Laboratory
Oak Ridge, TN, USA
{godoywf},{valerolarap},{teranishik},{pbalapra},{vetter}@ornl.gov

ABSTRACT

We evaluate the AI-assisted generative capabilities of OpenAI Codex on fundamental numerical kernels in high-performance computing (HPC): AXPY, GEMV, GEMM, SpMV, Jacobi Stencil, CG. We test the generated kernel codes for a variety of language-supported programming models: i) C++: OpenMP (including offload), OpenACC, Kokkos, SyCL, CUDA, HIP; ii) Fortran: OpenMP (including offload), OpenACC; iii) Python: numba, Numba, cuPy, pyCUDA, iv) Julia: Threads, CUDA.jl, AMDGPU.jl, KernelAbstractions.jl. We use GitHub Copilot capabilities available in Visual Studio Code, as of April 2023, to generate a vast amount of implementations given simple `<kernel> + <programming model> + <optional hints>` prompt variants. To quantify and compare the results, we propose a proficiency metric around the initial 10 suggestions given for each prompt. Results suggest that OpenAI Codex outputs in C++ correlate with the adoption and maturity of programming models: e.g. OpenMP, CUDA score really high, while HIP is still lacking. We found that prompts from either a targeted language like Fortran or the more general-purpose Python can benefit from adding code keywords, while Julia prompts perform acceptably well for its mature programming models: Threads and CUDA.jl. We expect that these benchmarks provide a point of reference for each programming model community. Overall, understanding the convergence of LLMs, AI and HPC is crucial due to its rapidly evolving nature in how it is redefining human-computer interactions.

KEYWORDS

OpenAI Codex; GPT; programming models; generative AI; numerical kernels; HPC; GitHub Copilot

1 INTRODUCTION

Since its initial release in 2020, the Generative Pre-trained Transformer 3 (GPT-3) [6] has signified a revolutionary step in the evolution of human-computer interactions. Developed by OpenAI[1], GPT-3 is the third generation prediction large language model (LLM) used for several AI generated human-like text applications. It has gained praise for the high-quality results in several natural language processing (NLP) [20] tasks, due in part to the unprecedented cost, US$12 million, and size of its training model of 175 billion parameters at 800 GB. Hence GPT-3, and its successor GPT-4 [2], are defining several societal questions for the near future.

As we enter the current era of exascale computing dominated by the extreme heterogeneity of our hardware and programming models [46], AI-assisted code generation could play a key role in how we develop, deploy, and test our software targeting high-performance computing (HPC) systems. Traditional human-readable code in programming languages used in HPC applications like C++ [42], Fortran [2], Python [45], and more recently Julia [4], are a straight-forward application for GPT-3 LLM capabilities that would help redefine software development. Hence, we need to understand the current state-of-practice, limitations and potential of this new technology.

We gather our early experiences in interacting with OpenAI Codex, a GPT-3 descendant, via the GitHub Copilot [3] plugin available on Visual Studio Code [4] for the generation

This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The publisher acknowledges the US government license to provide public access under the DOE Public Access Plan (https://energy.gov/downloads/doe-public-access-plan).

ICPP'23, August 2023, Salt Lake City, UT, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1]https://openai.com/
[2]https://openai.com/product/gpt-4
[3]https://github.com/features/copilot
[4]https://docs.github.com/en/copilot/getting-started-with-github-copilot?tool=vscode

of the implementation of relevant scientific kernels in HPC. Our goal is to have an initial assessment and gather an understanding of the impact of GPT-3 state-of-art capabilities in the overall interactive process for generating, optimizing, building and testing well-established mathematical HPC kernels. We evaluate i) AXPY, ii) general matrix-vector multiply, GEMV, iii) ii) the general matrix-matrix multiply, GEMM, iv) the Sparse matrix–vector multiplication, SpMV, v) 3D Jacobi stencil computations, and vi) Conjugate Gradients, CG. We test the generation of these kernels over a variety of programming models targeting CPU and graphical processing units (GPU) hardware. Our goal is to establish a benchmark and understand the status of "prompt" engineering in OpenAI Codex when applied to these important kernels and programming models as the technology continues to evolve rapidly.

The rest of the paper is structured as follows: Section 2 provides an overview of related efforts highlighting the recent attention to these topics in the broader area of Computer Science. We describe our prompt input pattern methodology for interacting with GitHub Copilot for the kernel generation process along with the proposed metric to quantify the quality of the suggested outputs in Section 3. Section 4 presents the results of our evaluation along with our findings on prompt trade-offs options for each language, kernel, programming model and additional keyword inputs on the generated outputs. We aim to understand the current status of the correctness, trade-offs and overall value of LLMs for HPC practitioners adopting this technology. Conclusions and future directions are presented in Section 5. Appendix A provides the artifact description for the reproducibility of this study.

2 BACKGROUND

The availability of GPT-3 has led to a recent trend in the literature focusing on the understanding of its large language model (LLM) capabilities for a large variety of applications. Brown et al. [6] introduced the seminal paper on GPT-3's unprecedented results showing strong performance on NLP interactions aspects such as translation, question-answering, and cloze tasks. They highlight the use and success rates of prompt-based [14, 15] "zero", "one", and "few" - shots learners (FSL) techniques when only few data examples are available to train a model as opposed to other previously trained machine learning (ML) alternatives. Wang et al. [47] present a comprehensive review on FSL pointing out that the "unreliable empirical risk minimizer" due to its nature to compensate for the lack of supervised information using prior available knowledge. Floridi and Chiriatti [17] provide a commentary on the nature of GPT-3, its scope and limits while outlining the social consequences of the "*industrialization of automatic cheap production of good, semantic artifacts*".

On the code generation side, Dehaerne et al. [10] provide a systematic review of the application of ML methods in description-to-code, code-to-code, and code-to-description studies from the past six years. They indicate limitations such as the variability in the quality of generated mined source code and the need and availability of large datasets, e.g. GitHub sources. In their work, it was highlighted that automatically generating data is a fast alternative for obtaining data but is only appropriate for certain contexts like "programming by example". Recent works using GitHub's Copilot "AI pair programmer", based on OpenAI Codex and the vast availability of source code hosted on GitHub, study GPT-powered products targeting specifically AI assisted code generation and programming. Chen et al. [8] provides an introduction and evaluation of Codex on Python code-writing capabilities. It is important to mention that Copilot uses a different Codex version from the one in this study. They point out the current limitations in the difficulty with "*docstrings describing long chains of operations and with binding operations to variables*". A major challenge noted is the over-reliance on Codex by novice programmers. Nguyen and Nadi [31] provide an empirical evaluation of GitHub Copilot suggestions for correctness and understandability to LeetCode questions in Java, JavaScript, Python, and C. Shortcomings included generating further simplified code that relies on undefined functions are discussed. Vaithilingam, Zhang and Glassman [43] provide a human subject study consisting of 24 participants on the usability of GitHub Copilot. They concluded that Copilot did not necessarily reduce the task completion time in common tasks such as file editing, web scrapping and graph plotting for experienced users of the Python language. Sobania et al. [41] found little difference between Copilot and automatic generators using Genetic Programming when applied to the PSB2 program synthesis benchmarks [19]. Imai [21] provides a preliminary assessment showing that while Copilot helps generate lines of code faster, the quality is lower when compared to human pair programming. Yetistiren et al. [49] assess the correctness, validity, and efficiency of targeting the HumanEval problem dataset [8] with a high success rate.

From an educational perspective, Sarsa et al. [40] discuss Codex's effectiveness in generating programming exercises and explanations via Copilot, not without addressing the need for oversight in output quality. Finnie-Ansley et al. [16] reports that Codex ranks high when compared to a class of introductory computer science students taking programming tests. Similarly, Denny et al. [11] discuss the pedagogical value of Copilot testing their answers and prompt nature in introductory programming questions. Wermelinger [48] discusses important questions for how teaching programming will evolve. Similarly, Brennan and Lesage [5] discuss the educational opportunities for undergraduate engineering and the need for students to have a strong intuition for software development when using AI-assisted tools.

Pierce et al. [37] assessed security aspects of Copilot's code contributions in a large sample resulting in relatively high vulnerability rates. They concluded that there is a need to quantify the limits of generated code in low-level scenarios and that Copilot must be paired with security-aware tooling.

Overall, there is a common narrative that the current state-of-the-art GPT-based tools are here to stay impacting almost every aspect of human-computer interactions. However, we are at an inflection point in which more exploratory research is needed when assessing AI code-generating capabilities and their technical, economic, and social implications need to be carefully understood.

3 METHODOLOGY

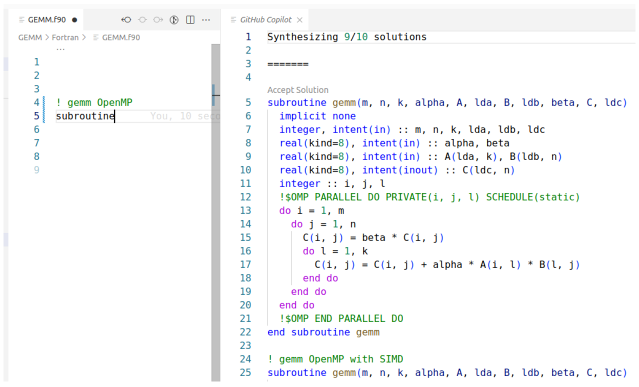
We evaluate prompt outputs for parallel programming models available in four different languages: C++, Fortran, Python, and Julia. Our methodology consists of two different aspects: i) Copilot code suggestion generation from prompt queries based on kernels, the parallel programming model for each language, and ii) defining and using a simple metric to evaluate the correctness of the results. As stated in their documentation[5], GitHub Copilot is *trained on all languages that appear in public repositories. For each language, the quality of suggestions you receive may depend on the volume and diversity of training data for that language.* Hence, we attempt to find out if these results correlate with the expected availability of correct programming models and public code examples. This perception is given by metrics of popularity, in particular in open source software [30], for i) number of repositories per language on GitHub, e.g. GitHut[6] or ii) the TIOBE index[7]. Hence, C++ and Python codes are expected to have wider availability than Julia and Fortran. Nevertheless, because of the general purpose nature of C++ and Python this metric might not be as relevant as for Fortran and Julia targeting scientific and mathematical applications.

We observe the outputs from each prompt and categorize them in a way we can measure their level of correctness. The rest of the section describes the experiment setup and the proposed metric to quantify the quality of the AI-generated results.

[5]https://docs.github.com/en/copilot/overview-of-github-copilot/about-github-copilot-for-individuals
[6]https://madnight.github.io/githut/#/pull_requests/2023/1
[7]https://www.tiobe.com/tiobe-index/

3.1 Experiments setup

We select a simple prompt base structure for each targeted language. Visual Studio Code recognizes the targeted language through the appropriate support plugin and the currently opened file based on language extension (e.g. cpp, f90, py and jl) and makes it part of the prompt prefix. The rest of the prompt is generated through a comment line complemented with optional code. Pressing `<Control> + <Enter>` results in a separate window containing a maximum of 10 code suggestions. Given the highly sensitive current nature of the LLM, even one additional keyword could heavily influence the outputs.



```
! gemm OpenMP
subroutine
```

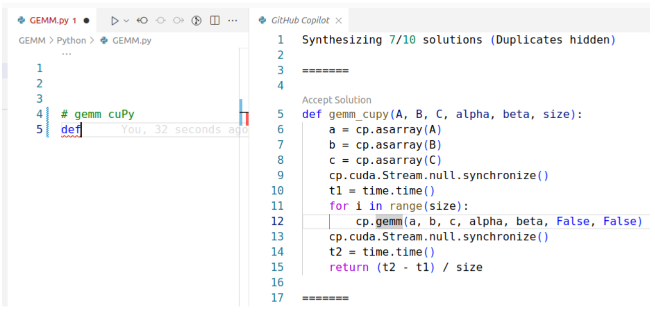
```
Synthesizing 9/10 solutions

=======

subroutine gemm(m, n, k, alpha, A, lda, B, ldb, beta, C, ldc)
  implicit none
  integer, intent(in) :: m, n, k, lda, ldb, ldc
  real(kind=8), intent(in) :: alpha, beta
  real(kind=8), intent(in) :: A(lda, k), B(ldb, n)
  real(kind=8), intent(inout) :: C(ldc, n)
  integer :: i, j, l
  !$OMP PARALLEL DO PRIVATE(i, j, l) SCHEDULE(static)
  do i = 1, m
    do j = 1, n
      C(i, j) = beta * C(i, j)
      do l = 1, k
        C(i, j) = C(i, j) + alpha * A(i, l) * B(l, j)
      end do
    end do
  end do
  !$OMP END PARALLEL DO
end subroutine gemm

! gemm OpenMP with SIMD
subroutine gemm(m, n, k, alpha, A, lda, B, ldb, beta, C, ldc)
```

(a)



```
# gemm cuPy
def
```

```
Synthesizing 7/10 solutions (Duplicates hidden)

=======

def gemm_cupy(A, B, C, alpha, beta, size):
    a = cp.asarray(A)
    b = cp.asarray(B)
    c = cp.asarray(C)
    cp.cuda.Stream.null.synchronize()
    t1 = time.time()
    for i in range(size):
        cp.gemm(a, b, c, alpha, beta, False, False)
    cp.cuda.Stream.null.synchronize()
    t2 = time.time()
    return (t2 - t1) / size

=======
```

(b)

Figure 1: GitHub Copilot prompt interface with optional language keywords for (a) Fortran's "subroutine" and (b) Python's "def".

Therefore our simple prompt queries can be described using the following structure:

- <kernel> <programming-model>
- <kernel> <programming-model> <optional keyword: function, subroutine, def>

Table 1 shows the programming language and model combinations used for this study. The additional post fix optional keywords correspond to our attempt to add more information to the prompt query to obtain better-quality suggestions. As it will be shown in Section 4, C++ models sensitivity can vary using the word “function" (not a language keyword), while Fortran and Python are consistently sensitive to the `subroutine` and `def` keywords. We also found that there is little sensitivity in Julia prompts when adding a postfix (e.g. “function"), hence we did not add it in this study.

Language	Programming Model	post fix
C++	OpenMP [36]	offload, function
	OpenACC [35]	function
	Kokkos [7]	function
	CUDA [33]	function
	HIP [1]	function
	Thrust [34]	function
Fortran	OpenMP	offload, subroutine
	OpenACC	subroutine
Python	numpy [44]	def
	Numba [29]	def
	pyCUDA [26]	def
	cuPy [32]	def
Julia	Threads [27]	
	CUDA [3]	
	AMDGPU [39]	
	KernelAbstractions [9]	

Table 1: Scope of our experimental setup applied for each kernel in terms of language and targeted parallel programming model

3.2 Correctness metric

To evaluate the correctness of the generated suggestions, we propose a simple approach that is based on our observations more than on any particular formalism. We consider five different levels of correctness and proficiency labels, between [0] or *non-knowledge* and [1] or *expert* when observing the suggested answers given by Copilot as those illustrated in Figure 1a and 1b for each combination in Table 1.

0 *non-knowledge*, no code at all or not a single correct code

.25 *novice*, one correct code, but includes other several correct or incorrect programming models (e.g. OpenACC suggestions in a OpenMP prompt)

.5 *learner*, one correct code, there are other incorrect codes, but all of them are using the requested programming model.

.75 *proficient*, all the codes are correct using the programming model requested.

1 *expert*, only one piece of code is provided and is totally correct.

4 RESULTS

To assess the accuracy and proficiency of the code suggestions generated by OpenAI Codex, we have developed a simple approach based on our observations rather than any specific formalism. We have categorized the correctness and proficiency of the suggestions into five levels, ranging from [0] or *non-knowledge* to [1] or *expert*. The suggested answers are analyzed based on these labels, as shown in Figure 1a and 1b, for each combination listed in Table1. This approach enables us to evaluate the effectiveness of OpenAI Codex in generating accurate and proficient code, providing valuable insights that can help users optimize their use of this technology.

4.1 C++

Table 2 shows the resulting metric for all of our C++ experiments, while Figure 2 illustrates the results according to the different kernels and programming models.

prompt	AXPY	GEMV	GEMM	SpMV	Jacobi	CG
prefix <kernel>						
OpenMP	.75	.5	.5	.5	0	.25
OpenMP offload	.5	.5	.5	.25	.25	0
OpenACC	.5	0	.25	0	0	0
Kokkos	.5	0	0	0	.25	0
CUDA	.75	.75	.75	0	0	.25
HIP	.75	0	0	0	.25	0
Thrust	.25	0	0	0	0	0
SyCL	.75	.25	0	0	0	0
prefix <kernel> postfix “function"						
OpenMP	.75	.75	.75	.25	.25	.25
OpenMP offload	.5	.5	.5	.25	.25	0
OpenACC	.5	.5	.5	.25	0	0
Kokkos	.75	.25	.25	0	.25	0
CUDA	.75	.25	0	0	0	0
HIP	.75	0	0	0	.25	0
Thrust	.5	0	.25	0	0	0
SyCL	.75	.5	.25	0	0	0

Table 2: Metric assessment for GitHub Copilot’s C++ outputs using the input prompt pattern “`<kernel> <programming model> (function)`"

As shown, the best results are achieved for the AXPY kernel. We see a clear trend in these results (Figure 2-left): the more complex the kernel the fewer quality results are obtained. While we see a level between *learner* and *proficient* for the AXPY kernel (the simplest kernel), for Conjugate Gradient (the most complicated kernel), the level is close to *non-knowledge*. Regarding programming models, we see

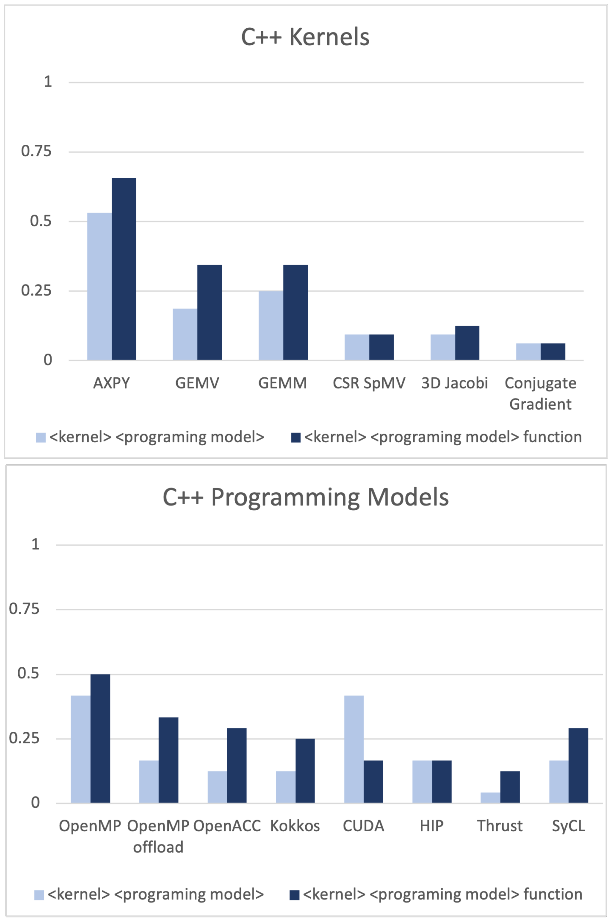


Figure 2: Results for C++ kernels (top) and programming models (bottom).

better results for OpenMP and CUDA. This could be due to the maturity of these programming models with respect to others and their availability in public code. We notice that the use of "function" as part of the prompt is very beneficial, which increments the quality of the results in the dense matrix cases. Sparse matrix and high level algorithms (SpMV, Jacobi and Conjugate Gradient) do not show much improvements. However, a different trend is observed for CUDA, where, instead of improving the level of proficiency, actually it decreases the quality of the answer. This is perhaps because the word "function" is not used for CUDA codes. In our experimentation - not shown here - the use of the words "kernel" or "`__global__`" lead to better code generation quality. This is an example of how important is to adapt the language used for the prompts to the particularities or syntax habitually used by such a community.

Results also show that high-level programming model prompts from Kokkos, Thrust, or SyCL perform poorly over several kernels. We view these results as a reflection of the user community size of these high-level abstractions. Over several instances, we observed that many wrong answers or no answers at all dominate as the kernel becomes more complex. It is important to point out the availability of large benchmark repositories such as HecBench [22, 23], from which some of the responses originate.

4.2 Fortran

Fortran is a particular case in this analysis due to its importance for the HPC and scientific community. Despite not being a "mainstream" language in terms of code availability, Copilot is able to provide some good results due to its domain-specific nature and legacy.

As seen in Table 3, the use of an "optimized" prompt using the "subroutine" keyword is particularly beneficial. Not using it leads to very poor results, with the AXPY OpenMP case being the only exception due to its simplicity and availability. We observe a similar trend as in C++, the more mature solutions like OpenMP and OpenACC give better results for parallel codes using Fortran.

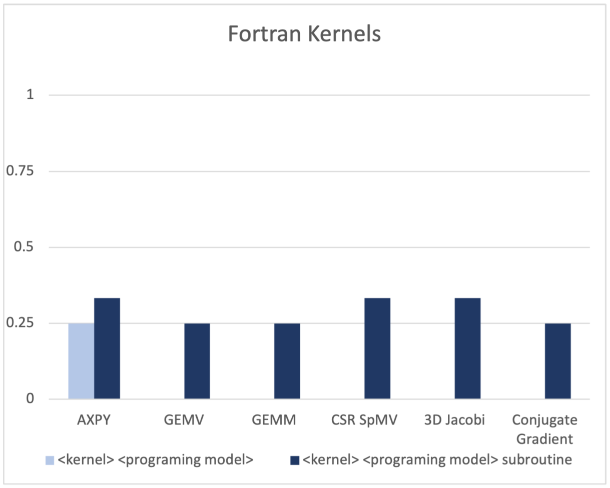
prompt	AXPY	GEMV	GEMM	SpMV	Jacobi	CG
prefix <kernel>						
OpenMP	.75	0	0	0	0	0
OpenMP offload	0	0	0	0	0	0
OpenACC	0	0	0	0	0	0
prefix <kernel> postfix "subroutine"						
OpenMP	.75	.25	.25	.5	.5	.25
OpenMP offload	.25	.25	.25	.25	.5	.25
OpenACC	.25	.25	.25	.25	.25	.25

Table 3: Metric assessment for Copilot's outputs using the input prompt pattern "`<kernel> <programming model> (subroutine)`" for Fortran

Figure 3 illustrates the uniform characteristic in the responses across kernels. Fortran has a lot of available legacy targeting HPC applications, so it is not surprising that we can obtain correct kernels implementations with higher complexity (CG, Jacobi) as well as more simple ones (AXPY, GEMM).

4.3 Python

Python is one of the most used general-purpose languages in industry, research, AI, and also has an important role for educational purposes and as a major target of AI-generative code as documented in Section 2. While neither a parallel programming model nor part of the standard, `numpy` is considered in our evaluation due to being the "de-facto" standard for scientific computing in Python.



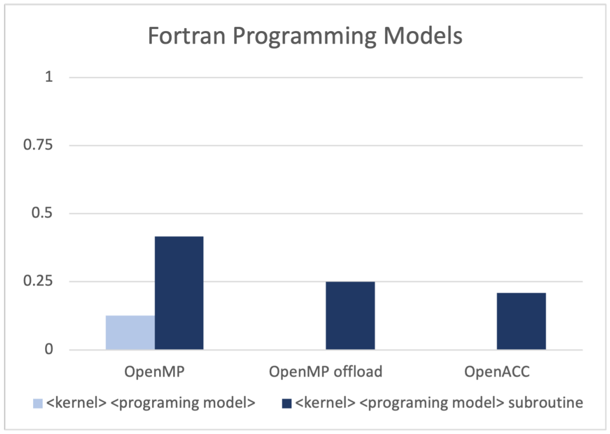


Figure 3: Results for Fortran kernels (top) and programming models (bottom).

Table 4 shows the resulting metric from our evaluation. Similar to Fortran, we observe that the quality improves dramatically with the addition of the Python `def` keyword to clarify the intention that we are looking for functions in the language. Overall, Copilot is able to generate acceptable numpy, cuPy and pyCUDA implementations across several kernels, while Numba falls behind. Perhaps this is an indication that the former is a more popular alternative in the community via lightweight layers on top of compiled C or CUDA code, rather than using Numba's just-in-time approach on top of LLVM. It's worth noticing that Numba deprecated support for AMD GPU hardware recently, hence reinforcing the vendor-specific target of GPU kernels written in Python favoring CUDA-like implementations. An interesting observation is that successful GPU (pyCUDA and cuPy) instances include a correct raw CUDA kernel source code as a user-defined kernel. Also, successful cuPy instances include a kernel source using cuPy's abstractions. These instances reflect the documentation of pyCUDA [25] and cuPy [38], respectively.

prompt	AXPY	GEMV	GEMM	SpMV	Jacobi	CG
prefix <kernel>						
numpy	.25	0	0	0	0	0
CuPy	0	0	.25	0	0	0
pyCUDA	0	0	0	0	0	0
Numba	0	0	0	0	0	0
prefix <kernel> postfix "def"						
numpy	.75	.25	.25	.5	.5	.75
CuPy	.5	.25	.25	.25	.25	.25
pyCUDA	.5	.25	.5	.5	.25	0
Numba	.25	0	0	0	0	0

Table 4: Metric assessment for Copilot's outputs using the input prompt pattern "`<kernel> <programming model> (def)`" for Python

As seen in Figure 4, the resulting kernels can have a varying degree of success, but most return at least one correct answer. This is perhaps attributed to the wide availability of Python and numpy code in public repositories. The trend also confirms the lack of Numba correct results and perhaps it is an opportunity to exploit its pure-Python nature. Although, as it was recently highlighted by Kailasa et al. [24] writing Numba code for complex algorithms can be as challenging as using a compiled language.

4.4 Julia

Due to its mathematical and performance focus, we included the Julia language in our experiments. Julia provides an interesting proposition for building a dynamic language on top of LLVM heavily influenced by Fortran in its syntax for targeting scientific computing problems. Julia provides an accessible programming model for CPU Threads, which is part of the base language, vendor-specific CUDA.jl and AMDGPU.jl, and KernelAbstraction.jl for portable kernels across vendors. In previous work [18] we showed promising results, not without gaps, for simple performance comparison on CPU and GPU runs.

Table 5 and Figure 5 show the metrics obtained for each kernel and programming model in our evaluation. For each programming model, we see a proficiency level between *novice* and *learner* for Threads (part of the base language) and CUDA.jl, which are the most used and mature programming models. AMDGPU.jl and KernelAbstractions.jl rank lower, which correlates with their novelty and the availability of example code targeting GPU hardware other than NVIDIA's.

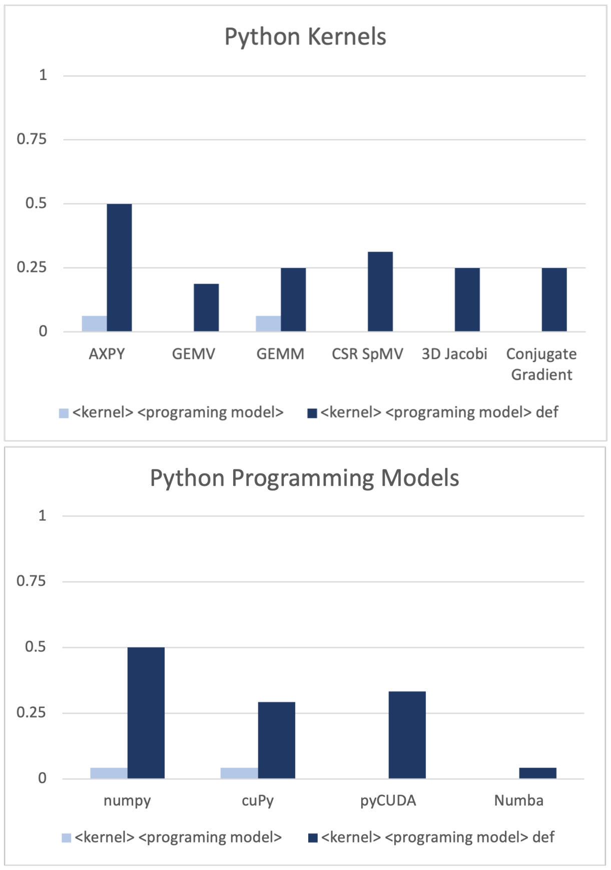


Figure 4: Results for Python kernels (top) and programming models (bottom).

Similarly to C++, a more complex kernel led to fewer correct results. For example, we could not get an appropriate implementation for the Conjugate Gradient kernel (a multi-kernel algorithm). Not surprisingly, as in the case of Fortran, Julia's mathematical nature allowed OpenAI Codex to find and suggest appropriate solutions in other kernels. The latter is despite Julia being a relatively new language with fewer publicly available codes, in particular, if compared to C++ or Python. We also note that our results did not necessarily improve by adding more information to the prompt (e.g. the `function` language keyword). This could be an advantage for domain-specific syntax, as in the case of Fortran, as most existing codes have a very targeted use with fewer words.

4.5 Discussion

To gain insights into the performance of OpenAI Codex across different languages and kernels, we have collated the results in Figure 6. As depicted in the graph, we observe similar trends to those observed in the previous graphs, where the complexity of the kernel directly impacts the quality of the results obtained. In other words, it becomes increasingly challenging to achieve acceptable results as the kernel's complexity increases.

We see a relatively "low" level of proficiency in OpenAI Codex with an average of *novice* level, for the languages and kernels tested. We see a slightly higher quality in the

prompt	AXPY	GEMV	GEMM	SpMV	Jacobi	CG
prefix <kernel>						
Threads	.75	.25	.5	0	0	0
CUDA	.75	.5	.5	.25	.25	0
AMDGPU	0	0	0	.25	0	0
KernelAbstractions	.25	.25	.25	.25	.25	0

Table 5: Metric assessment for Copilot's outputs using the input prompt pattern "`<kernel> <programming model>`" for Julia

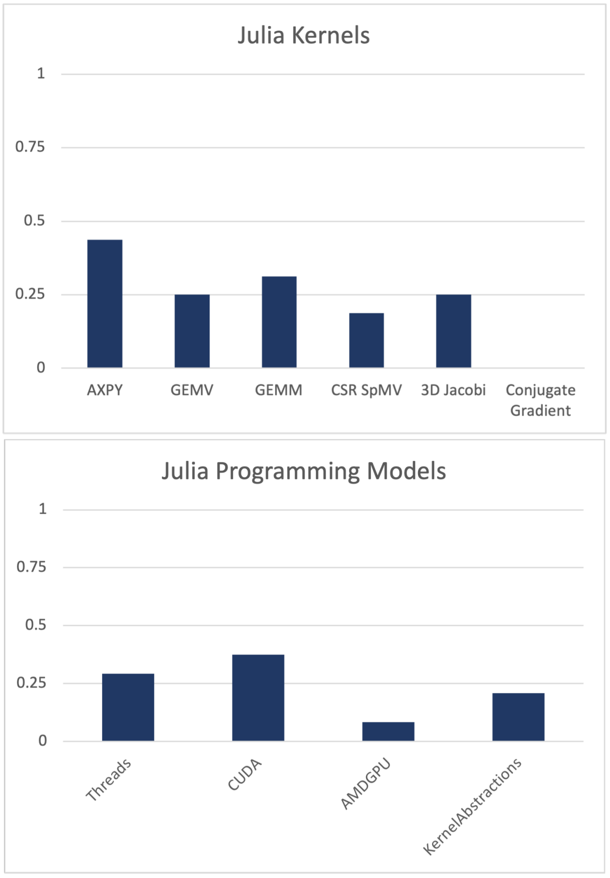


Figure 5: Results for Julia kernels (top) and programming models (bottom).

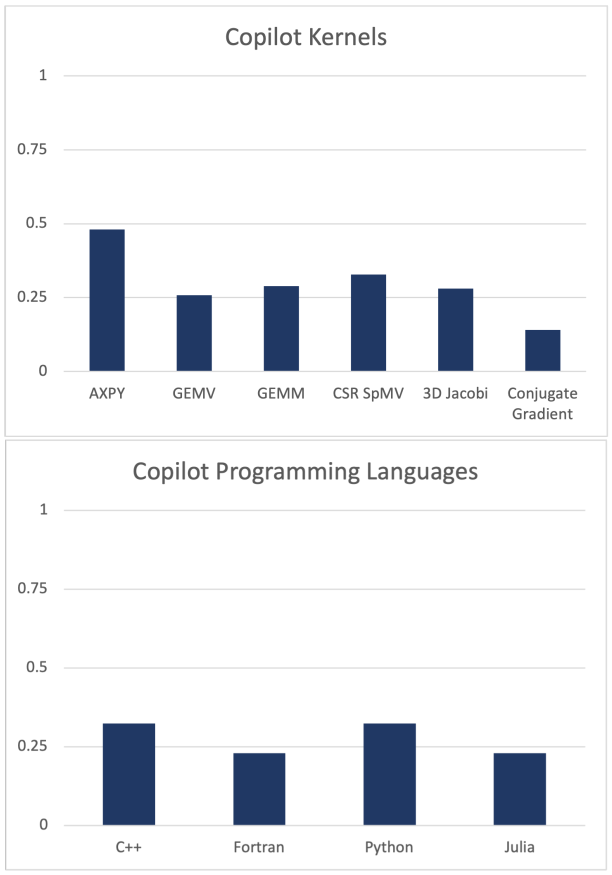


Figure 6: Overall results for Copilot kernels (top) and programming languages (bottom).

more popular C++ and Python languages than in Julia and Fortran. This may be related to the popularity, maturity, and accessibility of public codes implemented in those languages. However, languages like Fortran or Julia, provide very acceptable results. This confirms that it is more a question of targeted quality than of quantity in a specific domain as the scope of this work.

To summarize:

- As code complexity increases, obtaining acceptable results becomes more challenging.
- Generating high-quality multi-step or multi-kernel codes, such as Conjugate Gradient, can be difficult.
- The use of keywords can improve the proficiency of the answers, but it's essential to choose the correct words that are specific and sensitive to each programming language/model or community.
- While the popularity or accessibility of a programming language or public code can be important, less popular languages can also provide good results due to their targeted nature.

5 CONCLUSIONS AND FUTURE DIRECTIONS

We have carried out an initial study to evaluate the current capacity of OpenAI Codex via Copilot for the generation of HPC numerical kernels targeting parallel programming models in C++, Fortran, Python and Julia. Despite current limitations, we believe that generative AI can have an extraordinary beneficial impact on the HPC software development, maintenance and education in the future.

The research community still has several gaps to understand, and one such gap is the need for a more comprehensive taxonomy, akin to natural languages, to evaluate accuracy and trustworthiness. While our proposed taxonomy was beneficial for this initial study, it is imperative that such metrics be expanded to create a widely accepted methodology that the entire community can utilize. Therefore, there is a need for a standard and recognized approach to ensure uniformity in evaluating results across studies.

The emergence of LLMs technologies such as GPT-3 and other AI generative tools presents significant questions about their integration into the future ecosystem of HPC software development. For instance, can a human-in-the-loop and compiler be incorporated to refine initial LLMs suggestions? Can metadata-rich suggestions be incorporated to facilitate a human decision-making process? Additionally, how do significant HPC software modernization initiatives similar to DARPA's High Productivity Computing Systems (HPCS)[12] or the US Department of Energy Exascale Computing Project (ECP) [13, 28] incorporate these novel tools?

The automation of ecosystem aspects such as building systems, packaging, validation & verification, reproducibility, and continuous integration/continuous deployment (CI/CD) pipelines could significantly impact the HPC community. These technologies that put today's imperfect information closer to the human in question could redefine the educational aspects of HPC. Therefore, it is crucial to understand how each community can leverage these revolutionary capabilities to further advance their respective domains.

ACKNOWLEDGEMENT

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

REFERENCES

[1] AMD. 2022. AMD ROCm v5.2 Release. https://rocmdocs.amd.com/en/latest/Current_Release_Notes/Current-Release-Notes.html#amd-rocm-v5-2-release
[2] J. W. Backus and W. P. Heising. 1964. Fortran. *IEEE Transactions on Electronic Computers* EC-13, 4 (1964), 382–385. https://doi.org/10.1109/PGEC.1964.263818

[3] Tim Besard, Christophe Foket, and Bjorn De Sutter. 2018. Effective Extensible Programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems* (2018). https://doi.org/10.1109/TPDS.2018.2872064 arXiv:1712.03112 [cs.PL]

[4] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (Jan. 2017), 65–98. https://doi.org/10.1137/141000671 arXiv:http://dx.doi.org/10.1137/141000671

[5] Robert W. Brennan and Jonathan Lesage. 2023. Exploring the Implications of OpenAI Codex on Education for Industry 4.0. In *Service Oriented, Holonic and Multi-Agent Manufacturing Systems for Industry of the Future*, Theodor Borangiu, Damien Trentesaux, and Paulo Leitão (Eds.). Springer International Publishing, Cham, 254–266.

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[7] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202–3216. https://doi.org/10.1016/j.jpdc.2014.07.003 Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[9] Valentin Churavy, Dilum Aluthge, Lucas C Wilcox, James Schloss, Simon Byrne, Maciej Waruszewski, Julian Samaroo, Ali Ramadhan, Meredith, Simeon Schaub, Jake Bolewski, Anton Smirnov, Charles Kawczynski, Chris Hill, Jinguo Liu, Oliver Schulz, Oscar, Páll Haraldsson, Takafumi Arakaki, and Tim Besard. 2022. *JuliaGPU/KernelAbstractions.jl: v0.8.3.* https://doi.org/10.5281/zenodo.6742177

[10] Enrique Dehaerne, 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

[11] Paul Denny, Viraj Kumar, and Nasser Giacaman. 2023. Conversing with Copilot: Exploring Prompt Engineering for Solving CS1 Problems Using Natural Language. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (Toronto ON, Canada) *(SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1136–1142. https://doi.org/10.1145/3545945.3569823

[12] Jack Dongarra, Robert Graybill, William Harrod, Robert Lucas, Ewing Lusk, Piotr Luszczek, Janice Mcmahon, Allan Snavely, Jeffrey Vetter, Katherine Yelick, Sadaf Alam, Roy Campbell, Laura Carrington, Tzu-Yi Chen, Omid Khalili, Jeremy Meredith, and Mustafa Tikir. 2008. DARPA's HPCS Program: History, Models, Tools, Languages. In *Advances in COMPUTERS*. Advances in Computers, Vol. 72. Elsevier, 1–100. https://doi.org/10.1016/S0065-2458(08)00001-6

[13] Jack Dongarra et al. 2011. The International Exascale Software Project roadmap. *The International Journal of High Performance Computing Applications* 25, 1 (2011), 3–60. https://doi.org/10.1177/1094342010391989 arXiv:https://doi.org/10.1177/1094342010391989

[14] Li Fei-Fei, R. Fergus, and P. Perona. 2006. One-shot learning of object categories. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28, 4 (2006), 594–611. https://doi.org/10.1109/TPAMI.2006.79

[15] Michael Fink. 2004. Object Classification from a Single Example Utilizing Class Relevance Metrics. In *Advances in Neural Information Processing Systems*, L. Saul, Y. Weiss, and L. Bottou (Eds.), Vol. 17. MIT Press. https://proceedings.neurips.cc/paper_files/paper/2004/file/ef1e491a766ce3127556063d49bc2f98-Paper.pdf

[16] James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. 2022. The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming. In *Proceedings of the 24th Australasian Computing Education Conference* (Virtual Event, Australia) *(ACE '22)*. Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3511861.3511863

[17] Luciano Floridi and Massimo Chiriatti. 2020. GPT-3: Its nature, scope, limits, and consequences. *Minds and Machines* 30 (2020), 681–694.

[18] William F. Godoy, Pedro Valero-Lara, T. Elise Dettling, Christian Trefftz, Ian Jorquera, Thomas Sheehy, Ross G. Miller, Marc Gonzalez-Tallada, Jeffrey S. Vetter, and Valentin Churavy. 2023. Evaluating performance and portability of high-level programming models: Julia, Python/Numba, and Kokkos on exascale nodes. arXiv:2303.06195 [cs.DC]

[19] Thomas Helmuth and Peter Kelly. 2021. PSB2: The Second Program Synthesis Benchmark Suite. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Lille, France) *(GECCO '21)*. Association for Computing Machinery, New York, NY, USA, 785–794. https://doi.org/10.1145/3449639.3459285

[20] Julia Hirschberg and Christopher D. Manning. 2015. Advances in natural language processing. *Science* 349, 6245 (2015), 261–266. https://doi.org/10.1126/science.aaa8685 arXiv:https://www.science.org/doi/pdf/10.1126/science.aaa8685

[21] Saki Imai. 2022. Is GitHub Copilot a Substitute for Human Pair-Programming? An Empirical Study. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 319–321. https://doi.org/10.1145/3510454.3522684

[22] Zheming Jin. 2021. *The Rodinia Benchmarks in SYCL.* Technical Report. Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States).

[23] Zhemin Jin. 2023. Hecbench. https://github.com/zjin-lcf/HeCBench.

[24] Srinath Kailasa, Tingyu Wang, Lorena A. Barba, and Timo Betcke. 2023. PyExaFMM: an exercise in designing high-performance software with Python and Numba. arXiv:2303.08394 [cs.SE]

[25] Andreas Klöckner. [n. d.]. pycuda 2022.2.2 documentation. https://documen.tician.de/pycuda/. Accessed: 2023-04-20.

[26] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. 2012. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Comput.* 38, 3 (2012), 157–174. https://doi.org/10.1016/j.parco.2011.09.001

[27] Tobias Knopp. 2014. Experimental multi-threading support for the Julia programming language. In *2014 First Workshop for High Performance Technical Computing in Dynamic Languages*. IEEE, 1–5.

[28] Douglas Kothe, Stephen Lee, and Irene Qualters. 2019. Exascale Computing in the United States. *Computing in Science Engineering* 21, 1 (2019), 17–29. https://doi.org/10.1109/MCSE.2018.2875366

[29] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 1–6.

[30] Dongdong Lu, Jie Wu, Yongxiang Sheng, Peng Liu, and Mengmeng Yang. 2020. Analysis of the popularity of programming languages in open source software communities. In *2020 International Conference on Big Data and Social Sciences (ICBDSS)*. 111–114. https://doi.org/10.

1109/ICBDSS51270.2020.00033

[31] Nhan Nguyen and Sarah Nadi. 2022. An Empirical Evaluation of GitHub Copilot's Code Suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/3524842.3528470

[32] ROYUD Nishino and Shohei Hido Crissman Loomis. 2017. Cupy: A numpy-compatible library for nvidia gpu calculations. *31st confernce on neural information processing systems* 151, 7 (2017).

[33] NVIDIA. 2022. CUDA Toolkit Documentation - v11.7.0. https://developer.nvidia.com/cuda-toolkit

[34] NVIDIA. 2022. The API reference guide for Thrust, the CUDA C++ template library. https://docs.nvidia.com/cuda/thrust/index.html

[35] OpenACC Architecture Review Board. 2020. OpenACC Application Program Interface Version 3.1. https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.1-final.pdf

[36] OpenMP Architecture Review Board. 2021. OpenMP Application Program Interface Version 5.2. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

[37] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. 754–768. https://doi.org/10.1109/SP46214.2022.9833571

[38] Inc. Preferred Networks and Inc. Preferred Infrastructure. [n. d.]. CuPy – NumPy & SciPy for GPU. https://docs.cupy.dev/en/stable/. Accessed: 2023-04-20.

[39] Julian Samaroo, Valentin Churavy, Wiktor Phillips, Ali Ramadhan, Jason Barmparesos, Julia TagBot, Ludovic Räss, Michel Schanen, Tim Besard, Anton Smirnov, Takafumi Arakaki, Stephan Antholzer, Alessandro, Chris Elrod, Matin Raayai, and Tom Hu. 2022. *JuliaGPU/AMDGPU.jl: v0.4.1*. https://doi.org/10.5281/zenodo.6949520

[40] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models *(ICER '22)*. Association for Computing Machinery, New York, NY, USA, 27–43. https://doi.org/10.1145/3501385.3543957

[41] Dominik Sobania, Martin Briesch, and Franz Rothlauf. 2022. Choose Your Programming Copilot: A Comparison of the Program Synthesis Performance of Github Copilot and Genetic Programming. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Boston, Massachusetts) *(GECCO '22)*. Association for Computing Machinery, New York, NY, USA, 1019–1027. https://doi.org/10.1145/3512290.3528700

[42] Bjarne Stroustrup. 2013. *The C++ programming language*. Pearson Education.

[43] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI EA '22)*. Association for Computing Machinery, New York, NY, USA, Article 332, 7 pages. https://doi.org/10.1145/3491101.3519665

[44] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. 2011. The NumPy array: a structure for efficient numerical computation. *Computing in science & engineering* 13, 2 (2011), 22–30.

[45] Guido Van Rossum et al. 2007. Python Programming Language.. In *USENIX annual technical conference*, Vol. 41. Santa Clara, CA, 1–36.

[46] J. S. Vetter, R. Brightwell, M. Gokhale, P. McCormick, R. Ross, J. Shalf, K. Antypas, D. Donofrio, T. Humble, C. Schuman, B. Van Essen, S. Yoo, A. Aiken, D. Bernholdt, S. Byna, K. Cameron, F. Cappello, B. Chapman, A. Chien, M. Hall, R. Hartman-Baker, Z. Lan, M. Lang, J. Leidel, S. Li, R. Lucas, J. Mellor-Crummey, P. Peltz Jr., T. Peterka, M. Strout, and J. Wilke. 2018. *Extreme Heterogeneity 2018 - Productive Computational Science in the Era of Extreme Heterogeneity: Report for DOE ASCR Workshop on Extreme Heterogeneity*. Technical Report. USDOE Office of Science (SC) (United States). https://doi.org/10.2172/1473756

[47] Yaqing Wang, Quanming Yao, James T. Kwok, and Lionel M. Ni. 2020. Generalizing from a Few Examples: A Survey on Few-Shot Learning. *ACM Comput. Surv.* 53, 3, Article 63 (jun 2020), 34 pages. https://doi.org/10.1145/3386252

[48] Michel Wermelinger. 2023. Using GitHub Copilot to Solve Simple Programming Problems. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (Toronto ON, Canada) *(SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 172–178. https://doi.org/10.1145/3545945.3569830

[49] Burak Yetistiren, Isik Ozsoy, and Eray Tuzun. 2022. Assessing the Quality of GitHub Copilot's Code Generation. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering* (Singapore, Singapore) *(PROMISE 2022)*. Association for Computing Machinery, New York, NY, USA, 62–71. https://doi.org/10.1145/3558489.3559072

A ARTIFACT DESCRIPTION

The entire prompt input and resulting output sets are publicly available at: https://github.com/keitaTN/Copilot-hpc-kernels. Due to the rapid evolution and statistical nature of these technologies reproducibility and replicability of the present results is a challenging aspect that we expect to improve over time, but has no guarantees as of today. Hence, it's imperative that the reproducibility information is provided.

Some additional information:

- Experiments used the Visual Studio Code GitHub Copilot plugin service on three separate Linux systems using Ubuntu 22.04
- Experiments were carried between April 14th and April 21st of 2023.
- The raw dataset are identified by `<kernel>/<language>/<kernel>_outputs.<ext>` directory and file structure.
- Each Prompt used in this study is identified with the line comment `Prompt: <kernel> <programming model> <additional text>`.