

# A Directive-Based Programming Environment Using Code-Generating AI

Satoshi Ohshima

Research Institute for Information Technology, Kyushu University  
Fukuoka 819-0395, Japan  
ohshima@cc.kyushu-u.ac.jp

**Abstract.** Directive-based programming environments are widely used in high-performance computing. OpenMP and OpenACC are prominent examples, but many other directive-based language specifications have also been developed beyond these widely adopted standards. One challenge in this research is the cost, in effort and time, of developing language-processing systems such as compilers, translators, and frameworks. Although researchers can devise a directive specification and prepare test programs to evaluate its effectiveness, implementing a processing system and sustaining its development are not straightforward. This study therefore explores the use of rapidly advancing code-generating AI. We classify directive-based programming environments using code-generating AI into two categories: directive-processing systems that use code-generating AI directly (AI as Translator) and directive-processing systems implemented using code-generating AI (AI Generated Translator). We discuss their advantages and concerns and report on prototype implementations.

## 1. Introduction

A variety of programming languages, libraries, and frameworks are used to develop programs for today's high-performance computing (HPC) environments. Commonly used languages in HPC include C/C++, Fortran, and Python, with parallelization provided by OpenMP [1], OpenACC [2], CUDA [3], MPI [4], and other technologies. This study focuses specifically on directive-based programming environments, including languages or specifications and their implementations.

OpenMP and OpenACC are widely used directive-based programming environments in HPC. Other examples include the kernel loop directive in CUDA Fortran [5], the earlier High Performance Fortran [6], and various directive specifications developed primarily for research. Directives are also used to express compiler-specific features such as SIMD vectorization and cache control. Directive-based programming is a useful means of achieving both performance and productivity and is therefore an important research topic.

One major challenge in research on directive-based programming, particularly research that introduces a new language specification or extends an existing one, is the cost, in effort and time, of implementing a language-processing system such as a compiler or translator. Simple code substitutions may be easy to implement in a scripting language. However, implementation becomes costly when complex code rewriting is required or when the system must support a wide range of programs, including programs of practical complexity and scale. This cost can become a research bottleneck. Maintenance costs also create difficulties for sustaining the research and disseminating its results. The author has participated in several studies on directive-based programming [7], [8], [9], [10]. In some of these studies, the high implementation cost meant that directives were proposed and evaluated through manual source-code rewriting without implementing a processing system.

This study therefore considers directive-based programming environments that use rapidly advancing code-generating AI technology. If code-generating AI can undertake the costly implementation of language-processing systems, researchers should be able to concentrate their resources on the ideas behind the directives themselves and on evaluating their effectiveness.

The remainder of this paper is organized as follows. Section 2 reviews existing directive-based programming environments and related work. Section 3 presents the concept and classification of directive-based programming environments using code-generating AI. Section 4 describes prototype language-processing systems developed using code-generating AI and discusses their usefulness. Section 5 concludes the paper.

## 2. Directive-Based Programming Environments

This section provides an overview of existing directive-based programming environments, their implementations, and related research.

## 2.1 Existing Directive-Based Programming Environments

We first review existing directive-based programming environments, focusing primarily on HPC. Particular attention is paid to what is rewritten in the input program and at what level the rewriting takes place.

OpenMP and OpenACC are representative directive-based programming environments used in HPC.

OpenMP is a parallelization directive standard that enables thread-level parallelism by adding directives to C/C++ or Fortran programs. Many open-source and commercial compilers support OpenMP, and it is widely used in HPC because it can achieve high performance when used appropriately. Although thread-parallel loops are a representative use case, OpenMP supports various forms of thread-level parallelism, including non-loop parallel regions expressed with the sections directive and task parallelism expressed with the task directive. OpenMP also provides a mechanism for GPU parallelization, known as target offloading, and can thus be used broadly for parallelization within a process or a node. Among open-source compiler implementations, GCC [11] converts OpenMP source code into an intermediate representation and then into calls to runtime functions, with the libgomp library invoked at runtime. In LLVM [12], the Clang frontend parses the directives, the program is transformed at the intermediate-representation level, and the result is converted into runtime-function calls that invoke the libomp library at runtime. In contrast, the Omni OpenMP Compiler [13], developed in the Real World Computing Project (RWCP) and subsequently taken over by the University of Tsukuba and RIKEN, uses a frontend to convert the input into an intermediate representation, translates it back into thread-parallel C source code, and compiles that code with a native compiler.

OpenACC was standardized as a directive specification for GPU programming with an ease of use comparable to OpenMP. Owing in part to compiler support, however, it is currently used primarily for parallelization on NVIDIA GPUs. Like OpenMP, OpenACC makes loop parallelization straightforward and can often deliver reasonable performance for simple algorithms. Nevertheless, its expressiveness does not cover all GPU capabilities, so the cases in which OpenACC alone achieves sufficient performance are limited. As far as can be inferred from its observed behavior, NVIDIA's OpenACC compiler implementation translates the parallel regions specified by directives into PTX, an assembly-like intermediate representation for GPUs, and executes them through the CUDA Driver API. In contrast, the OpenACC extensions to the ROSE compiler developed at Lawrence Livermore National Laboratory [14], [15] and OpenARC developed at Oak Ridge National Laboratory [16] translate the input into their own intermediate representations, process the directives at that level, and then generate source code such as CUDA C/C++ for the target execution environment.

CUDA C/C++, which is essential for implementing high-performance programs for NVIDIA GPUs, is not a directive-based programming environment, but it does provide its own loop-unrolling directive, `#pragma unroll`. It also provides directives in the `#pragma nv_diag` family to control compiler behavior. Furthermore, CUDA Fortran includes a feature called the kernel loop directive, which, like OpenACC, allows GPU-parallel loops to be expressed easily using directives. This can be considered an advantage of CUDA Fortran over CUDA C/C++, which requires a separate function even for a simple GPU kernel. (CUDA C/C++ now also allows functions to be written in an inline-like manner using constructs such as lambda expressions.) Judging from its observed behavior, the CUDA Fortran kernel loop directive appears to be implemented similarly to OpenACC: the compiler translates the code into PTX, which is then executed through the CUDA Driver API.

High Performance Fortran (HPF), actively studied in the 1990s, is an example of a directive-based programming environment that attracted considerable attention in the past. HPF is a programming-language specification that extends Fortran 90 for distributed-memory environments. Directives such as `!HPF$ DISTRIBUTE` were used to declare data distribution, one of its key features. Various HPF implementations were attempted, apparently including both source-to-source compilers that translated programs into MPI programs or similar forms and native compilers.

Examples of directive-based programming environments used primarily for research include XcalableMP [17] and XcalableACC [18], developed mainly by research groups at the University of Tsukuba and RIKEN. XcalableMP is a partitioned global address space (PGAS) language that enables parallelization for distributed-memory environments by adding directives to sequential C or Fortran code. XcalableACC extends XcalableMP to computing environments equipped with accelerators such as GPUs. XcalableMP and XcalableACC inherit technologies from the Omni OpenMP Compiler mentioned above. They transform programs using a source-to-source infrastructure with XcodeML as the intermediate representation and then compile the results with a backend compiler.

ppOpen-AT [7] has been developed as a directive-based programming environment for autotuning. It can add autotuning capabilities to C and Fortran programs, making it easy, for example, to select an optimal loop-unrolling factor or to choose an optimal value from several candidates. ppOpen-AT is specified to perform code rewriting at the source-code level. The author has also participated in defining the ppOpen-AT specification, and the results of research on extensions for OpenACC [8] have been incorporated into both the specification and the implementation. The author has also been involved in research on optimizing arithmetic precision [9] and other topics, but those results have not been incorporated into the specification or implementation.

One of the author's earlier studies on directive-based programming environments was OMPCUDA [10], which generated CUDA programs from OpenMP programs before OpenACC and OpenMP target offloading were introduced. OMPCUDA rewrites programs with OpenMP loop parallelization into programs parallelized using CUDA C. Its implementation uses technologies from the Omni OpenMP Compiler to transform programs at the intermediate-representation level and generate CUDA C source code for GPU execution.

As these examples demonstrate, directive-based programming environments have a variety of specifications and implementations. Some are implemented as native compilers, whereas others perform only source-to-source translation and leave compilation to a native compiler. This study considers source-to-source translation.

## 2.2 Related Work on Programming Environments, Compilers, and AI

The application of AI to compilers has a long research history. The survey by Ashouri et al. [20] provides a detailed overview and history, while the survey by Rgaguena et al. [21] discusses recent applications of code-generating AI.

A well-known recent application of AI is Machine Learning Guided Optimization (MLGO) [22], introduced into LLVM by Google. This work applied reinforcement learning to inlining and register allocation in LLVM and demonstrated practical usefulness. Research on applying AI to compilers has continued to develop, with studies such as that of Cummins et al. [23] exploring the use of large language models (LLMs) for optimization within compilers.

Research is also progressing on AI-based source-code translation. TransCoder [24] used unsupervised machine learning to translate programs between C++, Java, and Python. SafeTrans [25] used LLMs to translate C programs into Rust. Such research is also considered important for maintaining legacy code and related purposes.

Prior work combining directives and AI includes the study by Pornmaneerattanatri et al. [26] on automatically generating OpenMP directives using LLMs.

## 3. Proposed Directive-Based Programming Environments Using Code-Generating AI

This study proposes directive-based programming environments using code-generating AI. This section classifies their architectures and discusses their advantages and disadvantages.

### 3.1 Rationale for Using Directives and Scope of This Study

Before discussing directive-based programming environments using code-generating AI in detail, we examine the significance and usefulness of directive-based programming itself and clarify the scope of this study.

We first consider the question of whether directives are necessary for appropriate code transformation if code-generating AI is sufficiently capable. In this study, we regard directives as an interface for making human intentions explicit and communicating information smoothly. We take the position that directive-based programming is useful as a tool for efficiently applying the knowledge of skilled and experienced practitioners. On the other hand, sufficiently advanced code-generating AI may eventually be able to generate excellent code that follows the user's intentions without directives. For example, Hoshino et al. [19] have attempted to optimize large-scale code using code-generating AI. Optimization of the large and complex programs studied in HPC, including programs executed on supercomputers, is thus beginning to appear realistic. Once code-generating AI acquires sufficient program-optimization capabilities, further discussion will be needed to determine whether it should be allowed some freedom in optimization, controlled through directives or similar mechanisms, or guided by another approach.

This paper considers only source-to-source translation that preserves human readability. It would also be possible to use code-generating AI for code rewriting without regard to readability or for some of the compilation process, but these possibilities are outside the scope of this study.

### 3.2 Classification Based on How Code-Generating AI Is Used

We divide directive-based programming environments using code-generating AI into two broad categories: directive-processing systems that use code-generating AI directly (AI as Translator; Fig. 1-A) and directive-processing systems implemented using code-generating AI (AI Generated Translator; Fig. 1-B).

#### A) AI as Translator



#### B) AI Generated Translator

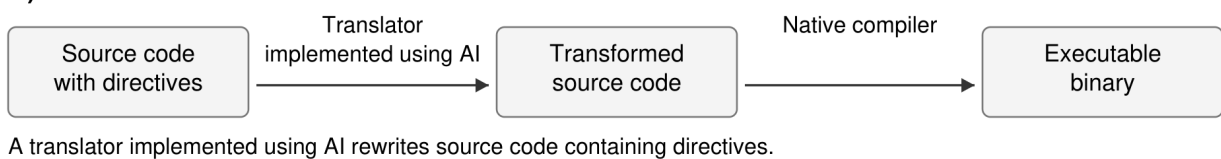


Figure 1. Classification of implementation approaches.

#### 3.2.1 A Directive-Processing System Using Code-Generating AI Directly (AI as Translator)

We first consider a directive-processing system that uses code-generating AI directly, referred to as AI as Translator (Fig. 1-A).

Code-generating AI can analyze and rewrite source code, and highly capable AI can perform complex rewrites. It therefore appears possible to use code-generating AI itself as the translator.

This approach has the following advantages. Unlike mechanical source-code translation, it allows the AI to exercise judgment during code transformation. Consequently, the AI may be able to detect the possibility that a rewrite will cause problems such as compilation errors or performance degradation. It may even be able to respond appropriately when the directive specification contains problems, such as overlooked cases.

This approach also has several disadvantages or concerns. First, the quality of source-code translation depends on the quality of the AI and how it is used. Simple code insertion or rewriting may not present a problem, but directive specifications involving complex code changes, or transformations of large and complex target programs, may require highly capable AI. In such cases, code transformation may take longer, raising concerns about increased usage fees for commercial LLMs or greater computing-resource requirements for local LLMs. If commercial LLMs are required, applying the approach to source code that cannot be disclosed externally may also be difficult. Furthermore, unlike a deterministic code-transformation tool, AI does not necessarily perform exactly the same processing every time it encounters the same task. The possibility of variation and uncertainty in the resulting source code must therefore also be considered.

#### 3.2.2 A Directive-Processing System Implemented Using Code-Generating AI (AI Generated Translator)

We next consider a directive-processing system implemented using code-generating AI, referred to as AI Generated Translator (Fig. 1-B).

In this approach, the role of code-generating AI is to implement a transformation tool that processes directives, rather than to rewrite source code directly. Its advantages and disadvantages are largely the reverse of those of the preceding approach, AI as Translator.

An advantage is that, once the translator has been implemented, it should transform code consistently each time. It is also easier to apply to highly confidential code. Because code-generating AI is not used for the

transformation itself, AI-service and GPU usage fees can be reduced. Processing times are also expected to be less likely to become excessive, even for large codebases.

On the other hand, this approach does not allow AI to exercise judgment during code transformation, increasing the need for a precise and complete directive specification and translator design. However, AI can exercise judgment while implementing the translator. We therefore expect that its completeness can be improved by asking the AI to check for deficiencies when reviewing the directive specification or implementing the translator, or by having it prepare test programs and verify the behavior.

## 4. Prototype Translators

In this section, we prototype directive-based programming environments using code-generating AI, examine their effectiveness, and discuss the results.

### 4.1 Prototyping a Simple Custom Directive Specification

For this experiment, we defined a simple custom directive specification and implemented both an AI as Translator system and an AI Generated Translator based on it. We then performed and evaluated code transformations.

#### 4.1.1 Problem Setting and Directive Specification

We designed the directive specification with loop optimization in mind. Although basic functionality needed by HPC programs is important, we considered it necessary to include somewhat more complex functionality in the tests as well. We therefore prepared the following four directives. Their names, basic syntax, and brief descriptions are given below. For the experiments, we prepared a specification document as a Markdown text file and provided it to the code-generating AI. The document included an overview of each directive; the meanings of its parameters, including their types and whether they were required; conditions governing its use; and one or two simple code transformation examples. The complete specification file, with some formatting changes for publication, is included in the appendix.

##### (1) Loop unrolling

```
#pragma xopt unroll(n)
```

Unrolls a loop by a factor of n.

##### (2) Switching between OpenMP and OpenACC

```
#pragma xopt autoacc(nOMP,nACC)
[(omp:option1)] [(acc:option2)]
```

Parallelizes the loop using OpenMP when the loop length is at least nOMP, or using OpenACC when it is at least nACC. Additional options for OpenMP and OpenACC can be specified in option1 and option2, respectively. Parameters enclosed in square brackets are optional; the same convention applies to the directives below.

##### (3) Hybrid OpenMP/OpenACC execution

```
#pragma xopt hybrid(rOMP,rACC)
[(omp:option1)] [(acc:option2)]
```

Divides the loop iterations in the ratio rOMP:rACC, parallelizing the rOMP portion on the CPU using OpenMP and the rACC portion on the GPU using OpenACC. Additional options for OpenMP and OpenACC can be specified in option1 and option2, respectively.

##### (4) Combining OpenMP/OpenACC switching with hybrid execution

```
#pragma xopt autohybrid(nOMP,nACC,nHYB,rOMP,rACC)
[(omp:option1)] [(acc:option2)]
```

Executes the loop sequentially when its length is less than nOMP; parallelizes it using OpenMP when its length is at least nOMP but less than nACC; and parallelizes it using OpenACC when its length is at least nACC but less than nHYB. When the loop length is at least nHYB, it divides the iterations in the ratio rOMP:rACC, parallelizing the rOMP portion on the CPU using OpenMP and the rACC portion on the GPU using OpenACC. Additional options for OpenMP and OpenACC can be specified in option1 and option2, respectively.

Specific transformation examples are given in the specification in the appendix. When the GPU is used, appropriate data transfers must be specified; the directive specification assumes that the necessary information

is supplied through `acc:option2`. However, as shown in the appendix, implementations of (3) and (4) use the `async` clause to perform computations on the CPU and GPU concurrently, which complicates data transfers between them. In this experiment, we intended to avoid this problem by using Managed Memory, but we did not provide this information when asking the code-generating AI to implement the translators. The AI as Translator case is discussed later.

#### 4.1.2 Experimental Setup

We used Claude Code and Codex in the experiments, operating both through their CLI clients on Ubuntu 24.04 LTS. For Claude Code, we used version 2.1.218 (Claude Code) with `claude-opus-4-8` (Opus 4.8), effort high. For Codex, we used `codex-cli 0.145.0` with `gpt-5.5 medium fast`. The programs were tested on a workstation equipped with an Intel Xeon W-2255 CPU and an NVIDIA T600. Because this experiment focused only on the correctness of the generated source code and whether basic functionality could be verified, execution time was not evaluated. We used `nvc 26.5` as the compiler, with the following compilation options:

```
-Minfo=accel -acc=gpu -gpu=ccnative,mem:managed -tp=host -mp
```

The test codes to be transformed were three simple numerical programs: vector addition, matrix multiply-add, and sparse matrix-vector multiplication with the matrix stored in compressed row storage (CRS) format. Figure 2 shows their computational kernels with the unrolling directive added.

```
// Vector addition
int vecadd
(const int N, double *c,
 const double *a, const double *b)
{
    int i;
    double d1, d2;
    d1 = omp_get_wtime();
#pragma xopt unroll(2)
    for(i=0; i<N; i++){
        c[i] = a[i] + b[i];
    }
    d2 = omp_get_wtime();
    printf("time: %lf sec\n", d2-d1);
    return 0;
}

// Matrix multiply-add
void matmat
(const int n, double *c,
 const double *a, const double *b)
{
    int i, j, k;
    double d1, d2;
    d1 = omp_get_wtime();
    for(i=0; i<n; i++){
        for(k=0; k<n; k++){
#pragma xopt unroll(2)
            for(j=0; j<n; j++){
                c[i*n+j] += a[i*n + k] * b[k*n+j];
            }
        }
    }
    d2 = omp_get_wtime();
    printf("time %lf sec\n", d2-d1);
}

// Sparse matrix-vector multiplication
void spmv
(const int n, const int nele,
 double *ans, const double *val,
 const int *irp, const int *icol,
 const double *vec)
{
    int i;
#pragma xopt unroll(2)
    for(i=0; i<n; i++){
        int j;
        double tmp = (double)0.0;
        for(j=irp[i]; j<irp[i+1]; j++){
            tmp += val[j] * vec[icol[j]];
        }
        ans[i] = tmp;
    }
}
```

Figure 2. Computational kernels of the test programs with the unrolling directive added.

#### 4.1.3 Experimental Results for AI as Translator

In the AI as Translator approach, the AI itself processes the code. Before providing the test codes and asking it to transform them, we therefore instructed each AI to prepare documentation for its own use so that it could efficiently rewrite code according to the directive specification.

We conducted the same experiment with Claude Code and Codex. We first placed a text file describing how to compile and run both the original and transformed sources, then issued the instructions shown in Figure 3. Both AIs transformed the source code, compiled it, ran tests, and reported the results. Inspection of the execution logs and generated artifacts confirmed that all codes had been transformed appropriately and could be executed. We did not use any data directives in the OpenACC-related codes, but no problems occurred because we

enabled Managed Memory through the compiler options. As these were small tests, the entire experiment took only a few minutes with either Claude Code or Codex.

Each directory contains several source files written using directives.  
 Each source file can be compiled and executed independently.  
 Process the directives in each source file and transform the code.  
 Name each transformed file by adding `_mod` to the original filename.  
 Example: `sample.c -> sample_mod.c`  
 Use the NVIDIA compiler to compile the programs.  
 It should already be available because the module has been loaded,  
 but run `module load nvhpc/26.5` if necessary.

*Figure 3. Instructions used to test the AI as Translator approach.*

One noteworthy observation was that Claude Code tested several additional parameter settings beyond the runtime parameters, such as problem sizes, that we had prepared. Both Claude Code and Codex also reported adjusting the code during transformation to avoid duplicate variable definitions. This issue had not been addressed explicitly in the directive specification. We thus unexpectedly confirmed the possibility discussed in Section 3.2.1: even when a directive specification has shortcomings, such as overlooked cases, AI judgment may enable an appropriate response. On the other hand, the specification for directive (2), switching between OpenMP and OpenACC, did not mention what should happen when the loop length was less than `nOMP`, and we had also omitted that branch from the code example. In response, Claude Code generated no processing for loop lengths below `nOMP`, whereas Codex generated code that executed the loop as is, without using either OpenMP or OpenACC. Although the Codex implementation matched our original intention, Claude Code's behavior was also reasonable given the information provided, making it difficult to determine which response was more appropriate.

#### 4.1.4 Experimental Results for AI Generated Translator

For the AI Generated Translator approach, we instructed each AI to implement a translator based on the specification document. The AIs asked some questions, including which implementation language to use, but we generally accepted their proposed default choices. We also instructed them to create their own test codes involving matrix and vector computations and to test their implementations. They performed tests using codes such as simple matrix multiplication and LU decomposition. These results support the observation in Section 3.2.2 that AI can help improve the completeness of an implementation.

The translator implemented by Claude Code was written in Python, with Python 3.6 or later specified as a requirement, and used only the Python standard library. It also documented restrictions on the input programs and limitations of the implementation, which should help with troubleshooting when code transformation fails.

The translator implemented by Codex was also written in Python; it specified Python 3 without giving a more precise version, and used only the Python standard library. Although it documented restrictions on input programs and implementation limitations, the documentation was less detailed than that produced by Claude Code. When we additionally instructed it to provide detailed information, it produced documentation comparable in detail to Claude Code's.

Both the Claude Code and Codex versions successfully transformed, compiled, and executed most of the codes. However, the omission in the specification for directive (2), switching between OpenMP and OpenACC, discussed in the previous subsection, affected this experiment as well. The Claude Code version omitted the branch for loop lengths below `nOMP` in all three test codes, whereas the Codex version included it in all three. In addition, when unrolling the sparse matrix-vector multiplication code, the Codex version generated duplicate local variable declarations, causing compilation to fail. The Claude Code version avoided this problem by enclosing each replicated code block in braces (`{` and `}`). Furthermore, the logs from the previous subsection showed that the issue had been detected and corrected during code transformation. It is therefore likely that this problem could have been resolved by running more thorough tests during translator implementation or by providing information about it as a common code transformation pitfall.

## 4.2 Prototyping a Translator Compatible with ppOpen-AT

As a more advanced experiment in prototyping translators using code-generating AI, we attempted to have code-generating AI implement a directive translator for ppOpen-AT.

ppOpen-AT is distributed under the MIT License through a GitHub repository (<https://github.com/Post-PetaCrest/ppOpenHPC/tree/AT/>). In addition to the specification in PDF form and sample codes, the translator source code is available. The distributed translator is implemented in C. It does not use an intermediate representation, external libraries, or parser generators such as yacc/lex.

In this experiment, we cloned ppOpen-AT from the repository, removed the translator, and asked the AIs to implement one using the remaining resources. We used the same Claude Code and Codex as in the preceding section and instructed them to refer to the available resources when implementing the translator. They again asked some questions, including which implementation language to use, but we generally accepted their proposed default choices. We did not specify the order in which features should be implemented. Because the purpose was prototyping rather than a complete implementation of every ppOpen-AT feature, we stopped each AI after it had made some implementation progress and asked a question. We instructed each AI to leave documentation at the end, which resulted in extensive records of implementation choices and procedures for continuing development of features that remained unimplemented.

We first examine the implementation produced by Claude Code. According to the records, Claude Code appears to have spent approximately eight hours on implementation. This excludes time spent waiting for responses to questions and refers to the time during which Claude Code was actually working. Claude Code selected Python 3.9 or later as the implementation language and implemented a broad range of features. However, its records stated that it had prioritized fixed-form Fortran because many of the samples originally included with ppOpen-AT used that form. Several limitations remained, including support for loop nests only up to a depth of three and the lack of support for multiple input files. Code analysis and rewriting were implemented using regular expressions, without external libraries or an intermediate representation.

We next examine the implementation produced by Codex. According to the records, Codex appears to have spent approximately four hours on implementation. Again, this excludes time spent waiting for responses to questions and refers to the time during which Codex was actually working. Codex selected Python 3.10 or later as the implementation language and implemented a broad range of features. It listed support for complex loop structures and C pointer arrays among the unimplemented items. As with Claude Code, code analysis and rewriting were implemented using regular expressions, without external libraries or an intermediate representation.

For functional checks, we performed code transformation and compilation on the 19 samples included with ppOpen-AT: three in C and 16 in Fortran 90. Some samples encountered transformation or compilation problems even with the original ppOpen-AT because they assumed older compilers or were incompatible with libraries installed in the experimental environment. We therefore used codes in which Claude Code had corrected these issues, limiting the changes to those that could be made in the samples without modifying ppOpen-AT. The original version successfully transformed 17 samples and compiled 16. The implementation produced by Claude Code successfully compiled 14 samples after transformation, whereas the implementation produced by Codex successfully compiled all 19 after transformation. However, neither the Claude Code version nor the Codex version implemented every feature, and some outputs did not contain the appropriate code changes. This qualification is important when interpreting the results. In the first place, some internal details of ppOpen-AT's code transformation process are not defined by its specification. In addition to rewriting source code containing directives, ppOpen-AT creates several additional source files according to those directives. In both the Claude Code and Codex versions, we found some cases in which the rewritten original source code exactly matched the output of the original implementation. However, the newly generated source code allowed flexibility in formatting and naming conventions. As a result, for every sample, differences were observed among the original version, the Claude Code version, and the Codex version. These results underscore the importance of thorough code review and functional testing.

## 5. Conclusion

There is a demand for directive-based programming environments and research on them in HPC, but the cost of implementing and maintaining their translators is a major issue. In this study, we proposed using rapidly advancing code-generating AI to create such systems, namely code transformation tools or translators, and classified and prototyped the approaches. The two categories are AI as Translator, in which code-generating AI itself processes directives, and AI Generated Translator, in which a directive translator is implemented using code-generating AI. We summarized the advantages and concerns associated with each. We then devised a simple custom directive specification and prototyped translators using both approaches. Evaluation of each

category with Claude Code and Codex showed that the systems behaved largely as intended and that both approaches could benefit from AI assistance, for example in addressing shortcomings in a directive specification. We also observed cases in which Claude Code and Codex produced different implementations when the specification left room for interpretation. As a more advanced prototyping experiment, we asked code-generating AI to implement a ppOpen-AT directive translator. Because these implementations were only partial, they did not achieve sufficient quality. Nevertheless, the experiment showed that the portions of source code generated from scratch differed in coding style, so the implementations did not produce identical code. This result further indicates the importance of review and functional testing when implementing directive translators with AI.

Much remains to be discussed and investigated experimentally. The directive specifications used in our prototypes were limited to simple transformations of a single loop; complex cases involving global code changes remain unexplored. Moreover, although all of the prototype directives were simple enough to illustrate with straightforward implementation examples, overlooked cases still caused Claude Code and Codex to produce different implementations. Open questions include how to reduce variation arising from AI judgment; how to present specifications efficiently for features whose implementation examples are themselves difficult or time-consuming to prepare; and whether the approach can handle features for which an implementation method is not obvious. One possible approach is to introduce a process in which the directive specification is refined through dialogue with AI before implementation. How to carry out such a process efficiently is itself an open question. Further discussion of the more fundamental question of the usefulness of directives themselves would also be valuable.

## Acknowledgments

This work was supported by JST's research and development program for next-generation edge AI semiconductors (JPMJES2511), JSPS KAKENHI (JP26K23819), and the Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures (JHPCN), project jh260071.

## References

- [1] OpenMP, <https://www.openmp.org/> (Accessed 2026-08-28)
- [2] OpenACC, <https://www.openacc.org/> (Accessed 2026-08-28)
- [3] CUDA Programming Guide, <https://docs.nvidia.com/cuda/cuda-programming-guide/index.html> (Accessed 2026-08-28)
- [4] MPI Forum, <https://www.mpi-forum.org/> (Accessed 2026-08-28)
- [5] CUDA Fortran, Kernel Loop Directive, <https://docs.nvidia.com/hpc-sdk/compiler/cuda-fortran-prog-guide/index.html#kernel-loop-directive>
- [6] Andreas Müller, Roland Rühl, "Extending high performance Fortran for the support of unstructured computations", In Proceedings of the 9th international conference on Supercomputing (ICS '95), Association for Computing Machinery, pp.127-136, 1995, <https://doi.org/10.1145/224538.224552>
- [7] Takahiro Katagiri, "ppOpen-AT: A Directive-base Auto-tuning Language", arXiv, 2024, <https://arxiv.org/abs/2408.16607>
- [8] Satoshi Ohshima, Takahiro Katagiri, Masaharu Matsumoto, "Utilization and Expansion of ppOpen-AT for OpenACC", 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp.1496-1505, 2016, <https://doi.org/10.1109/IPDPSW.2016.123>
- [9] Shouhei Yamanashi, Hisashi Yashiro, Takahiro Katagiri, Toru Nagai, Satoshi Ohshima, "Autotuning Power Consumption and Computation Accuracy using ppOpen-AT", 2022 IEEE 15th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MC-SoC), pp.208-215, 2022, <https://doi.org/10.1109/MCSoC57363.2022.00041>
- [10] Satoshi Ohshima, Shoichi Hirasawa, Hiroki Honda, "OMPCUDA : OpenMP Execution Framework for CUDA Based on Omni OpenMP Compiler", Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More, Springer, pp.161-173, 2010, [https://doi.org/10.1007/978-3-642-13217-9\\_13](https://doi.org/10.1007/978-3-642-13217-9_13)
- [11] GCC, <https://gcc.gnu.org/> (Accessed 2026-08-28)
- [12] LLVM, <https://llvm.org/> (Accessed 2026-08-28)
- [13] Omni OpenMP Compiler Project, <https://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/> (Accessed 2026-08-28)
- [14] ROSE Compiler, <https://computing.llnl.gov/projects/rose-compiler> (Accessed 2026-08-28)
- [15] Tristan Vanderbruggen, John Cavazos, "Generating OpenCL C kernels from OpenACC", In Proceedings of the International Workshop on OpenCL 2013 & 2014 (IWOCL'14), Association for Computing Machinery, pp.1-10, 2014, <https://doi.org/10.1145/2664666.2664675>
- [16] Lee Seyong, Jeffrey S. Vetter, "OpenARC: Extensible OpenACC Compiler Framework for Directive-Based Accelerator Programming Study", 2014 First Workshop on Accelerator Programming using Directives, pp.1-11, 2014, <https://doi.org/10.1109/WACCPD.2014.7>

- [17] Jinpil Lee, Mitsuhsa Sato, "Implementation and Performance Evaluation of XcalableMP: A Parallel Programming Language for Distributed Memory Systems", The 39th international Conference on Parallel Processing Workshops (ICPPW10), pp.413-420, 2010, <https://doi.org/10.1109/ICPPW.2010.62>
- [18] Masahiro Nakao, Hitoshi Murai, Takenori Shimosaka, Akihiro Tabuchi, Toshihiro Hanawa, Yuetsu Kodama, Taisuke Boku, Mitsuhsa Sato, "XcalableACC: Extension of XcalableMP PGAS Language using OpenACC for Accelerator Clusters", Workshop on accelerator programming using directives (WACCPD), pp.27-36, 2014, <https://doi.org/10.1109/WACCPD.2014.6>
- [19] Tetsuya Hoshino, Shun-Ichiro Hayashi, Daichi Mukunoki, Takahiro Katagiri, Toshihiro Hanawa, "Evaluating Claude Code's Coding and Test Automation for GPU Acceleration of a Legacy Fortran Application: A GeoFEM Case Study", Proc. the Supercomputing Asia and International Conference on High Performance Computing in Asia Pacific Region Workshops (SCA/HPCAsiaWS '26) - The 1st International Workshop on Foundational Large Language Models Advances for HPC in Asia (LLM4HPCAsia 2026), Association for Computing Machinery, pp.353-360, 2026, <https://doi.org/10.1145/3784828.3785335>
- [20] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, Cristina Silvano, "A Survey on Compiler Autotuning using Machine Learning", ACM Computing Surveys, Vol.51, No.5, pp.1-42, 2018, <https://doi.org/10.1145/3197978>
- [21] Aymane Rgaguena, Imane Chlioui, Maryam Radgui, "Generative AI for Code Translation: A Systematic Mapping Study", Engineering Proceedings, Vol.112, No.33, pp.1-9, 2025, <https://doi.org/10.3390/engproc2025112033>
- [22] Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, David Li, "MLGO: a Machine Learning Guided Compiler Optimizations Framework", arXiv, 2021, <https://arxiv.org/abs/2101.04808>
- [23] Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, Hugh Leather, "Large Language Models for Compiler Optimization", arXiv, 2023, <https://arxiv.org/abs/2309.07062>
- [24] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanut, Guillaume Lample, "Unsupervised translation of programming languages", In Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS '20), Curran Associates Inc., Article No.1730, pp.20601-20611, 2020,
- [25] Muhammad Farrukh, Baris Coskun, Tapti Palit, Michalis Polychronakis, "SafeTrans: LLM-assisted Transpilation from C to Rust", In Proceedings of the 1st Workshop on Code Translation, Transformation, and Modernization (ReCode '26), Association for Computing Machinery, pp.30-37, 2026, <https://doi.org/10.1145/3786180.3788317>
- [26] Soratouch Pornmaneerattanatri, Keichi Takahashi, Yutaro Kashiwa, Kohei Ichikawa, Hajimu Iida, "Toward Automatic Parallelization: Detecting Parallelizable Loops Using Large Language Modeling", Journal of Information Processing Systems Vol.21, No.3, pp.271-283, 2025, <https://doi.org/10.3745/JIPS.04.0349>

## Appendix

### A.1 Directive Specification Used in the Prototype

#### Provisional Directive Specification (xopt Directive Framework)

##### Loop Unrolling

###### Description

```
#pragma xopt unroll(n)
```

- n: integer ( $n > 0$ ).

###### Semantics

- Unroll the loop by a factor of n.
- Required option: the unrolling factor n (a positive integer).

###### Conditions

- The directive may be used only immediately before a for loop.
- There must be no dependencies between loop iterations. Ensuring this is the responsibility of the programmer.

##### Code Translation Examples

###### Before translation 1

```
#pragma xopt unroll(2)
for(i=0; i<N; i++){
    a[i] = b[i] + c[i];
}
```

###### After translation 1

```
// unrolled loop
for(i=0; i+1<N; i+=2){
    a[i] = b[i] + c[i];
    a[i+1] = b[i+1] + c[i+1];
}
// remainder loop
for(; i<N; i++){
    a[i] = b[i] + c[i];
}
```

###### Before translation 2

```
#pragma xopt unroll(2)
for(int i=0; i<N; i++){
    a[i] = b[i] + c[i];
}
```

###### After translation 2

```
int _xopt_i;
// unrolled loop
for(_xopt_i=0; _xopt_i+1<N; _xopt_i+=2){
    a[_xopt_i] = b[_xopt_i] + c[_xopt_i];
    a[_xopt_i+1] = b[_xopt_i+1] + c[_xopt_i+1];
}
// remainder loop
for(; _xopt_i<N; _xopt_i++){
    a[_xopt_i] = b[_xopt_i] + c[_xopt_i];
}
```

##### OpenMP/OpenACC Switching

###### Description

```
#pragma xopt autoacc(nOMP, nACC) [(omp:option1)] [(acc:option2)]
```

- nOMP, nACC: integers ( $nOMP > 0$ ,  $nACC > 0$ ,  $nOMP < nACC$ ).
- option1, option2: strings.

###### Semantics

- Use OpenMP parallelization when the loop length is at least nOMP, and OpenACC parallelization when it is at least nACC. Assume  $nOMP < nACC$ .
- Required options: the loop-length thresholds nOMP and nACC.
- Optional settings: option1 may specify OpenMP clauses (such as schedule), and option2 may specify OpenACC clauses (such as data directives).

**Conditions**

- The directive may be used only immediately before a for loop.
- There must be no dependencies between loop iterations. Ensuring this is the responsibility of the programmer.
- The loop structure must support parallelization with both OpenMP and OpenACC. Ensuring this is the responsibility of the programmer. The programmer must also account for data transfers between the CPU and GPU.

**Code Translation Example****Before translation 1**

```
#pragma xopt autoacc(nomp,nacc) (omp:schedule(dynamic)) (acc:independent)
for(i=0; i<N; i++){
    a[i] = b[i] + c[i];
}
```

**After translation 1**

```
if(N>=nacc){
    #pragma acc kernels loop independent
    for(i=0; i<N; i++){
        a[i] = b[i] + c[i];
    }
}else if(N>=nomp){
    #pragma omp parallel for schedule(dynamic)
    for(i=0; i<N; i++){
        a[i] = b[i] + c[i];
    }
}
```

**OpenMP/OpenACC Hybrid Execution****Description**

```
#pragma xopt hybrid(rOMP,rACC) [(omp:option1)] [(acc:option2)]
```

- rOMP, rACC: integers (rOMP > 0, rACC > 0).
- option1, option2: strings.

**Semantics**

- Partition the loop iterations in the ratio rOMP:rACC. Parallelize the rOMP portion on the CPU using OpenMP and the rACC portion on the GPU using OpenACC.
- Required options: the partitioning weights rOMP and rACC.
- Optional settings: option1 may specify OpenMP clauses (such as schedule), and option2 may specify OpenACC clauses (such as data directives).

**Conditions**

- The directive may be used only immediately before a for loop.
- There must be no dependencies between loop iterations. Ensuring this is the responsibility of the programmer.
- The loop structure must support parallelization with both OpenMP and OpenACC. Ensuring this is the responsibility of the programmer. The programmer must also account for data transfers between the CPU and GPU.

**Code Translation Example****Before translation 1**

```
#pragma xopt hybrid(romp,racc) (omp:schedule(dynamic)) (acc:independent)
for(i=0; i<N; i++){
    a[i] = b[i] + c[i];
}
```

**After translation 1**

```
int _xopt_split = (int)((double)romp/(romp+racc)*N);
#pragma acc kernels loop independent async
for(i=_xopt_split; i<N; i++){
    a[i] = b[i] + c[i];
}
#pragma omp parallel for schedule(dynamic)
for(i=0; i<_xopt_split; i++){
    a[i] = b[i] + c[i];
}
#pragma acc wait
```

## Combining OpenMP/OpenACC Switching and Hybrid Execution

### Description

```
#pragma xopt autohybrid(nOMP,nACC,nHYB,rOMP,rACC) [(omp:option1)] [(acc:option2)]
```

- nOMP, nACC, nHYB, rOMP, rACC: integers (nOMP > 0, nACC > 0, nHYB > 0, nOMP < nACC, nACC < nHYB).
- option1, option2: strings.

### Semantics

- Execute sequentially when the loop length is less than nOMP. Use OpenMP parallelization when the loop length is at least nOMP but less than nACC, and OpenACC parallelization when it is at least nACC but less than nHYB. When the loop length is at least nHYB, partition the loop iterations in the ratio rOMP:rACC, parallelizing the rOMP portion on the CPU using OpenMP and the rACC portion on the GPU using OpenACC.
- Required options: the thresholds nOMP, nACC, and nHYB, and the partitioning weights rOMP and rACC.
- Optional settings: option1 may specify OpenMP clauses (such as schedule), and option2 may specify OpenACC clauses (such as data directives).

### Conditions

- The directive may be used only immediately before a for loop.
- There must be no dependencies between loop iterations. Ensuring this is the responsibility of the programmer.
- The loop structure must support parallelization with both OpenMP and OpenACC. Ensuring this is the responsibility of the programmer. The programmer must also account for data transfers between the CPU and GPU.

### Code Translation Example

#### Before translation 1

```
#pragma xopt autohybrid(nomp,nacc,nhyb,romp,racc) (omp:schedule(dynamic)) (acc:independent)
for(i=0; i<N; i++){
    a[i] = b[i] + c[i];
}
```

#### After translation 1

```
if(N>=nhyb){
    int _xopt_split = (int)((double)romp/(romp+racc)*N);
    #pragma acc kernels loop independent async
    for(i=_xopt_split; i<N; i++){
        a[i] = b[i] + c[i];
    }
    #pragma omp parallel for schedule(dynamic)
    for(i=0; i<_xopt_split; i++){
        a[i] = b[i] + c[i];
    }
    #pragma acc wait
}else if(N>=nacc){
    #pragma acc kernels loop independent
    for(i=0; i<N; i++){
        a[i] = b[i] + c[i];
    }
}else if(N>=nomp){
    #pragma omp parallel for schedule(dynamic)
    for(i=0; i<N; i++){
        a[i] = b[i] + c[i];
    }
}else{
    for(i=0; i<N; i++){
        a[i] = b[i] + c[i];
    }
}
```