Codee Training:
Write Accelerated Code at Expert Level

Codee: Automated Code Inspection for Modernization and Optimization

NERSC Codee Training Series

September 5-6, 2024

Schedule

Day 1 (Thursday 5th, 9:00 - 12:30 PDT)

Codee: Automated Code Inspection for Modernization and Optimization

- Lecture:
  - *Codee's command-line tool*
  - *Open Catalog of Best Practices for Fortran/C/C++ Modernization and Optimization for CPU and GPU*
- Demo using Fortran:
  - *HIMENO modernization*
  - *HIMENO optimization through GPU parallelism*
- Demo using C/C++:
  - *MATMUL optimization through CPU parallelism*
- Hands-on: PI, MATMUL, COULOMB, HIMENO

Day 2 (Friday 6th, 9:00 - 12:30 PDT)

Codee: Automated Analysis of Large-Scale Fortran/C/C++ Codes

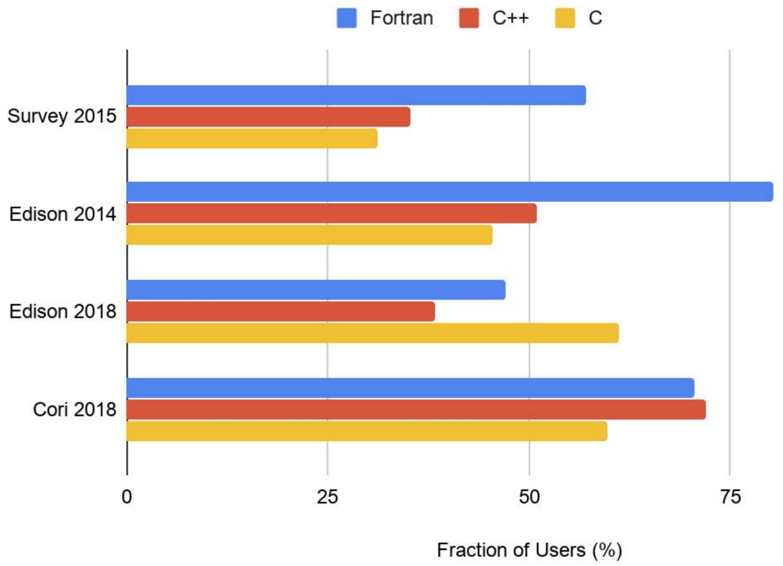
- Lecture:
  - *Codee's command-line tool using compilation databases*
  - *Automated testing of large codes using Codee on Perlmutter*
  - *Use case: Optimizing the Weather Research and Forecasting Model with OpenMP Offload and Codee*
- Demo using Fortran:
  - *Putting it all together with HYCOM*
- Demo using C/C++:
  - *Putting it all together with MBedTLS*
- Hands-on: HYCOM, NUCCOR, ATMUX, LULESHmk, MBedTLS
- Bring your own applications!

Your Main Drivers for Simulation Software?

- Simulation software demands **high-speed computations** and **maintainable code**.
- **<u>1. Modernize your code:</u>**
  - Adopt modern programming practices to increase code **quality** and **facilitate maintenance**:
    - Update legacy code; e.g.: F77 → F2018, C++98 → C++20.
    - Ensure portability across compilers; no vendor-specific language extensions.
    - Leverage new language features; e.g.: Fortran modules, C++ smart pointers.
  - The modernization process helps **find bugs** and **avoid introducing hidden bugs** during maintenance.
  - As a result, the modernization process helps ensure code **correctness**.
- **<u>2. Optimize performance</u>**:
  - Overall, **enforce modernization before addressing performance optimization**.

Fortran/C/C++ on NERSC

- Widely used in:
  - Aerospace
  - Automotive
  - Climate & Weather
  - Defense
  - Energy & Utilities
  - High Performance Computing
  - Manufacturing
  - Oil and Gas
  - Scientific Research
- Employed by **most NERSC users**.



Source: https://portal.nersc.gov/project/m888/nersc10/workload/N10_Workload_Analysis.latest.pdf

Language	First appeared in…	TIOBE Index 2024
Fortran	1957	10th
C	1972	3rd
C++	1985	2nd

codee

Thoughts of the **Fortran community** on modernization

“Always use IMPLICIT NONE everywhere. **It is amazing how many bugs this can find and avoid** compared to the default typing rules.”

“All subprograms should be CONTAINed. Generally in modules, but also in the main program unit. <...> **Again, amazing how many interface bugs show up when this is enforced**.”

“**Many many more could be suggested.** Here are a few in no specific order that **help compilers find more bugs at compile time, and help programs scale better**:

- Always specify intent attributes for dummy arguments.
- Always use assumed shape for array dummy arguments.”

“Always use Standard conforming code. **Turn on all warnings** (e.g., -std=f2018 -Wall with gfortran) and fix any issues by using Standard conforming code. There are really very few compiler extensions from the Olden Days that do not have modern, Standard conforming, replacements.”

Source:
https://fortran-lang.discourse.group/t/our-initiative-to-publish-the-fortran-lang-top-10-recommendation-for-fortran-modernization-is-it-really-new-or-even-feasible/7774/18

Top 10 Recommendations for Fortran Modernization

1. Strict compliance with modern Fortran standards

Remove legacy features deleted from recent Fortran standards, as they might not be supported by recent compilers, and avoid compiler-specific extensions, ensuring that the code remains compatible across various development environments.

2. Declare procedures in modules

Declare related procedures within an importable module to enhance code modularity, reusability, and readability. This practice also helps avoid runtime errors related to implicit interfaces. Separate the definition of procedures into modules and their implementation into submodules, leveraging incremental compilation to reduce build times.

3. Restrict data visibility with modules

Encapsulate globally accessible data, such as common blocks, within modules. This approach allows for controlled access interfaces, improving code readability and minimizing side effects from global data storage.

4. Improve dummy arguments semantics

Enhance the definitions of dummy arguments to make the behavior of procedures more predictable and transparent, helping avoid common issues related to incorrect assumptions about data type, flow, or structure.

5. Improve data type consistency and management

Ensure the consistency of data types by avoiding implicit typing and standardizing the code on a fixed set of real kinds, improving readability and portability across different development environments. Use derived data types to represent complex multi-field structures. Leverage pointers and allocatable arrays for safer memory handling.

6. Avoid legacy control-flow constructs

Replace outdated and error-prone control-flow constructs with more robust and maintainable language features from recent standards, improving code maintainability and reducing the likelihood of bugs.

7. Enhance source code semantics

Leverage elements from recent Fortran standards to further improve the clarity and intent of code statements beyond previous recommendations.

8. Adherence to code conventions

Establish and adhere to a consistent coding standard, such as variable naming rules of free-form format, to promote readability and ease collaboration among developers.

9. Adopt modern development practices

Integrate modern development practices, such as automated testing, version control, or dependency managers, to enhance the quality, maintainability, collaboration, and distribution of Fortran software.

10. Proper C/C++ interoperability

Ensure seamless interoperability between Fortran and C/C++ to allow Fortran programs to effectively interact with a wide range of systems and libraries written in other languages (e.g., high-performance environments).

github.com/codee-com/fortran-modernization

USE CASE: FORTRAN MODERNIZATION

Top 20 Checks for Fortran Modernization

- [M01] Tune compiler flags to mark non-standard and removed features in modern Fortran standards.
- [M01] Consider using more standard-compliant compilers like gfortran to flag non-standard and removed features.
- [M01] Consider replacing GNU Fortran non-standard constructs to favor portability
- [M02] PWR068: Encapsulate external procedures within modules to avoid the risks of calling implicit interfaces.
- [M03] PWR073: Transform common block into a module for better data encapsulation.
- [M03] PWR069: Use the keyword only to explicitly state what to import from a module.
- [M04] PWR008: Declare the intent for each procedure argument.
- [M04] PWR070: Declare array dummy arguments as assumed-shape arrays.
- [M05] PWR007: Always use implicit none to disable implicit declarations.
- [M05] PWR071: Prefer real(kind=kind_value) for declaring consistent floating types.
- [M06] PWR063: Avoid using legacy and old-style Fortran constructs.
- [M07] PWR003: Explicitly declare pure functions.
- [M07] Add an explicit parameter attribute to constant variables.
- [M07] PWR072: Add an explicit save attribute when initializing variables in their declaration.
- [M10] Consider using Fortran modules instead of C/C++ header files

github.com/codee-com/open-catalog

Open Catalog of Best Practices for Modernization and Optimization

Check		Fortran	C	C++	Autofix
Modernization					
PWR007	Disable implicit declaration of variables	✓			✓
PWR068	Encapsulate external procedures within modules to avoid the risks of calling implicit interfaces	✓			
PWR070	Declare array dummy arguments as assumed-shape arrays	✓			
Many more!					
Optimization					
PWR050	Consider applying multithreading parallelism to forall loop	✓	✓	✓	✓
PWR055	Consider applying offloading parallelism to forall loop	✓	✓	✓	✓
PWD006	Missing deep copy of non-contiguous data to the GPU	✓	✓	✓	
Many more!					

github.com/codee-com/open-catalog

Codee: Value Proposition

WHAT	HOW	WHERE
Community-guided Top Recommendations	**Open Catalog on Best Practices**	codee
First **Top 10 Recommendations for Fortran Modernization** published on February 2024	Containing **80+ checks** as of August 2024, with **curated documentation and examples**	**Automated analysis** of Codee 2024.3 reported **64.603 checks in WRF** running on Perlmutter

USE CASE: FORTRAN MODERNIZATION

Success Stories using Open Source Software

Code	Domain	Metrics with Codee 2024.3.0 (Aug. 2024)
CP2K 1.3M lines of code	Quantum chemistry and solid state physics software package	1344 files, 5431 functions, 9549 loops successfully analyzed and 17 non-analyzed files in 13 m 33 s
OpenRadioss 1.1M lines of code	Finite element solver for dynamic event analysis	3477 files, 6541 functions, 39636 loops successfully analyzed and 0 non-analyzed files in 31 m 6 s
WRF 960K lines of code	Weather Research and Forecasting	508 files, 9722 functions, 26519 loops successfully analyzed (64603 checkers) and 0 non-analyzed files in 1 h 17 m 18 s
ICON 646K lines of code	Weather, climate, and environmental prediction	1143 files, 6959 functions, 7801 loops successfully analyzed (6098 checkers) and 7 non-analyzed files in 9 m 34 s
SIESTA 398K lines of code	First-principles Materials Simulation	967 files, 2956 functions, 2254 loops successfully analyzed (3291 checkers) and 25 non-analyzed files in 2 m 16 s
PHASTA 64K lines of code	Parallel Hierarchic Adaptive Stabilized Transient Analysis of compressible and incompressible Navier Stokes equations	284 files, 608 functions, 1086 loops successfully analyzed (1420 checkers) and 0 non-analyzed files in 6 m 35 s
HYCOM 44K lines of code	HYbrid Coordinate Ocean Model	50 files, 251 functions, 2058 loops successfully analyzed (2965 checkers) and 0 non-analyzed files in 53.85 s
EAP-patterns 4K lines of code	Patterns from an Eulerian cell AMR application	12 files, 88 functions, 164 loops successfully analyzed and 0 non-analyzed files in 1037 ms

Codee: Main Features

Static Analysis

Automatically analyze every line of code to find and fix modernization and optimization opportunities.

Code Coverage

Obtain code coverage metrics and discover lines that are not being analyzed.

Reports

Get a deeper understanding of your code's health with analysis reports.

Autofix

Automatically generate fixes for opportunities, always under the control of the programmer and preserving 100% code correctness.

CI/CD automation

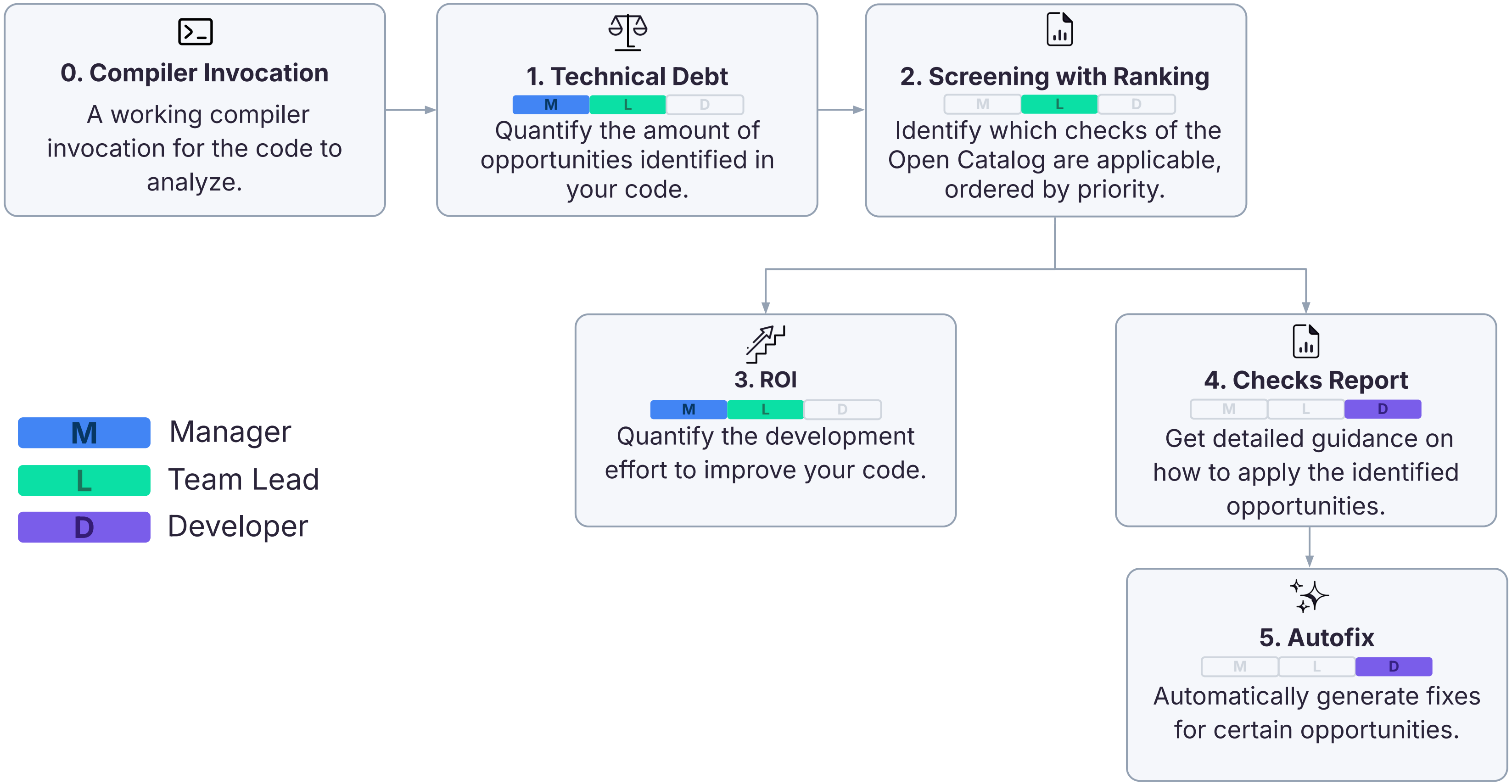
Integrate with CI/CD systems, automatically testing every code change and pull request.

Self-hosting

Execution on the local system, retraining full control of your code and privacy.

> **Codee provides a systematic, predictable workflow that is a complement to other software development tools, such as the compiler, profiler, or debugging tools.**

Codee: Suggested Basic Workflow



0. Compiler Invocation

```
$ cat -n matmul.f90
1    subroutine matmul(n, A, B, C)
2      double precision, dimension(n, n), intent(in) :: A, B
3      double precision, dimension(n, n), intent(out) :: C
4
5      ! Initialization
6      do i = 1, n
7        do j = 1, n
8          C(i, j) = 0.0
9        end do
10     end do
11
12     ! Accumulation
13     do i = 1, n
14       do j = 1, n
15         do k = 1, n
16           C(i, j) = C(i, j) + A(i, k) * B(k, j)
17         end do
18       end do
19     end do
20   end subroutine matmul

$ gfortran matmul.f90
```

1. Technical Debt Report

M L D

```
$ codee technical-debt -- gfortran matmul.f90

TECHNICAL DEBT REPORT

This report quantifies the technical debt associated with the modernization of legacy code by assessing the
extent of refactoring required for language constructs. The score is determined based on the number of
language constructs necessitating refactoring to bring the source code up to modern standards.
Additionally, the metric identifies the impacted source code segments, detailing affected files, functions,
and loops.

Score Affected files Affected functions Affected loops
----- -------------- ------------------ --------------
12    1              1                  4

TECHNICAL DEBT BREAKDOWN

Lines of code Analysis time Checkers Technical debt score
------------- ------------- -------- --------------------
16            12 ms         12       12

1 file, 1 function, 5 loops successfully analyzed (12 checkers) and 0 non-analyzed files in 12 ms
```

Score and affected source code

2. Screening with Ranking Report

M | L | D

```
$ codee screening -- gfortran matmul.f90

SCREENING REPORT

Lines of code Analysis time # checks Profiling
------------- ------------- -------- ---------
16            13 ms         12       n/a

RANKING OF CHECKERS

Checker Level Priority # Title
------- ----- -------- - ---------------------------------------------------------------------------------------
PWR039  L1    P27      1 Consider loop interchange to improve the locality of reference and enable vectorization
PWR068  L1    P27      1 Encapsulate external procedures within modules to avoid the risks of calling implicit
                         interfaces
RMK015  L1    P27      1 Tune compiler optimization flags to increase the speed of the code
PWR003  L1    P18      1 Explicitly declare pure functions
PWR008  L1    P18      1 Declare the intent for each procedure parameter
PWR070  L1    P18      1 Declare array dummy arguments as assumed-shape arrays
PWR071  L2    P6       2 Prefer real(kind=kind_value) for declaring consistent floating types
PWR007  L2    P6       1 Disable implicit declaration of variables
PWR035  L3    P2       1 Avoid non-consecutive array access to improve performance
RMK010  L3    P0       2 The vectorization model states the loop is not a SIMD opportunity due to strided memory
                         accesses
```

Total checks triggered

Checks ordered by priority

SUGGESTED BASIC WORKFLOW

3. ROI Report

M L D

```
$ codee roi -- gfortran matmul.f90

ROI ANALYSIS SUMMARY

This analysis underscores the tangible benefits Codee brings to the development process, not only in terms of savings
in development effort, but also in realizing significant cost efficiencies for the organization.

Impact on Development Effort:
This report identifies critical areas within the source code that necessitate attention from the development team, and
forecasts a significant reduction in workload by an estimated 223 hours.

Without Codee | With Codee | Hours saved
------------- | ---------- | -----------
235 hours     | 12 hours   | 223 hours

Impact on Cost Savings:
Considering a standard developer's workload of approximately 1800 hours/year, Codee's intervention translates to saving
an equivalent to 0.12 (223h / 1800h) developers working full-time. Assuming an average cost of a developer for the
company (salary + associated costs) of €100,000, this amounts to cost savings of €12,388 (€100,000 x 0.12).

Developer hours/year | Number of devs. saved/year | Developer salary/year | Total costs saved/year
-------------------- | -------------------------- | --------------------- | ----------------------
1800 hours           | 0.12                       | €100,000              | €12,388

ROI CALCULATION BREAKDOWN

<...>
```

Saved hours

Saved cost

4. Checks Report

M L D

```
$ codee checks -- gfortran matmul.f90

CHECKS REPORT

matmul.f90:6:3 [PWR039] (level: L1): Consider loop interchange to improve the locality of reference and enable vectorization
matmul.f90:1:1 [PWR068] (level: L1): Encapsulate external procedures within modules to avoid the risks of calling implicit interfaces
matmul.f90 [RMK015] (level: L1): Tune compiler optimization flags to increase the speed of the code
matmul.f90:1:1 [PWR003] (level: L1): Explicitly declare pure functions
matmul.f90:1:1 [PWR008] (level: L1): Declare the intent for each procedure parameter
matmul.f90:1:1 [PWR070] (level: L1): Declare array dummy arguments as assumed-shape arrays
matmul.f90:1:1 [PWR007] (level: L2): Disable implicit declaration of variables
matmul.f90:2:3 [PWR071] (level: L2): Prefer real(kind=kind_value) for declaring consistent floating types
matmul.f90:3:3 [PWR071] (level: L2): Prefer real(kind=kind_value) for declaring consistent floating types
matmul.f90:13:3 [PWR035] (level: L3): Avoid non-consecutive array access to improve performance
matmul.f90:7:5 [RMK010] (level: L3): The vectorization model states the loop is not a SIMD opportunity due to strided memory accesses
matmul.f90:15:7 [RMK010] (level: L3): The vectorization model states the loop is not a SIMD opportunity due to strided memory accesses
```

Checks by location

```
$ codee checks --verbose -- gfortran matmul.f90

CHECKS REPORT

<...>

matmul.f90:1:1 [PWR007] (level: L2): Disable implicit declaration of variables
  Suggestion: Add IMPLICIT NONE in the specification part of the procedure 'matmul'
  Documentation: https://github.com/codee-com/open-catalog/tree/main/Checks/PWR007
  AutoFix:
    codee rewrite --modernization implicit-none --in-place matmul.f90:matmul -- gfortran matmul.f90

<...>
```

Detailed information on each check

5. Autofix

M L D

```
$ codee rewrite --modernization implicit-none --in-place matmul.f90:matmul -- gfortran matmul.f90

Results for file '/home/user/matmul.f90':
  Successfully applied AutoFix to the procedure at 'matmul.f90:1:1' [using insert implicit none]:
      [INFO] Inserted implicit none:
        - matmul.f90:1:1

Successfully updated matmul.f90

$ git diff matmul.f90

 subroutine matmul(n, A, B, C)
+  ! Codee: Made all variable declarations explicit (2024-08-02 13:04:38)
+  implicit none
+  integer :: i
+  integer :: j
+  integer :: k
+  integer :: n
   double precision, dimension(n, n), intent(in) :: A, B
   double precision, dimension(n, n), intent(out) :: C
```

Focus the Analysis: Select a subset of checks (I)

Common options:

`--check-id <id>[,<id>]*`
Focus on specific checks

`--target-arch <arch>`
Focus on multiple checks

`--list-available-checkers`
List all available checks

Focus the Analysis: Select a subset of checks (II)

```
$ codee checks --target-arch cpu,gpu -- gfortran matmul.f90

<...>

CHECKS REPORT

matmul.f90:6:3 [PWR039] (level: L1): Consider loop interchange to improve the locality of reference and enable
vectorization
matmul.f90:1:1 [PWR068] (level: L1): Encapsulate external procedures within modules to avoid the risks of calling
implicit interfaces
matmul.f90 [RMK015] (level: L1): Tune compiler optimization flags to increase the speed of the code
matmul.f90:1:1 [PWR003] (level: L1): Explicitly declare pure functions
matmul.f90:1:1 [PWR008] (level: L1): Declare the intent for each procedure parameter
matmul.f90:1:1 [PWR070] (level: L1): Declare array dummy arguments as assumed-shape arrays
matmul.f90:1:1 [PWR007] (level: L2): Disable implicit declaration of variables
matmul.f90:13:3 [PWR050] (level: L2): Consider applying multithreading parallelism to forall loop
matmul.f90:2:3 [PWR071] (level: L2): Prefer real(kind=kind_value) for declaring consistent floating types
matmul.f90:3:3 [PWR071] (level: L2): Prefer real(kind=kind_value) for declaring consistent floating types
matmul.f90:13:3 [PWR055] (level: L3): Consider applying offloading parallelism to forall loop
matmul.f90:13:3 [PWR035] (level: L3): Avoid non-consecutive array access to improve performance
matmul.f90:7:5 [RMK010] (level: L3): The vectorization cost model states the loop is not a SIMD opportunity due to
strided memory...
matmul.f90:15:7 [RMK010] (level: L3): The vectorization cost model states the loop is not a SIMD opportunity due to
strided memory...
```

Focus the Analysis: Select a subset of checks (III)

```
$ codee checks --verbose --target-arch cpu,gpu -- gfortran matmul.f90

CHECKS REPORT

<...>

matmul.f90:13:3 [PWR055] (level: L3): Consider applying offloading parallelism to forall loop
  Suggestion: Use 'rewrite' to automatically optimize the code
  Documentation: https://github.com/codee-com/open-catalog/tree/main/Checks/PWR055
  AutoFix (choose one option):
    * Using OpenMP (recommended):
      codee rewrite --offload omp-teams --in-place matmul.f90:13:3 -- gfortran matmul.f90
    * Using OpenACC:
      codee rewrite --offload acc --in-place matmul.f90:13:3 -- gfortran matmul.f90
    * Using OpenMP and OpenACC combined:
      codee rewrite --offload omp-teams,acc --in-place matmul.f90:13:3 -- gfortran matmul.f90

<...>
```

Focus the Analysis: Select a subset of checks (IV)

```
$ codee rewrite --offload omp-teams --in-place matmul.f90:13:3 -- gfortran matmul.f90

Results for file 'matmul.f90':
  Successfully applied AutoFix to the loop at 'matmul.f90:matmul:13:3' [using offloading]:
      [INFO] matmul.f90:13:3 Parallel forall: variable 'C'
      [INFO] matmul.f90:13:3 Loop parallelized with teams using OpenMP directive 'target teams distribute parallel for'
  Fine-tuning suggestions for better performance [using offloading]:
      [TODO] Consider optimizing data transfers of arrays by adding the proper array ranges in data mapping clauses
          Documentation: https://github.com/codee-com/open-catalog/tree/main/Glossary/Offloading-data-transfers.md

$ git diff matmul.f90

   ! Accumulation
+  ! Codee: Loop modified by Codee (2024-08-02 13:35:36)
+  ! Codee: Technique applied: offloading with 'omp-teams' pragmas
+  ! Codee: Offloaded loop: begin
+  ! TODO (Codee): Consider optimizing data transfers of arrays by adding the proper array ranges in data mapping
clauses
+  !$omp target teams distribute parallel do shared(A, B, n) map(to: n, A, B) private(j) map(tofrom: C)
schedule(static)
   do i = 1, n
     do j = 1, n
       do k = 1, n
@@ -17,4 +22,5 @@ subroutine matmul(n, A, B, C)
       end do
     end do
   end do
+  ! Codee: Offloaded loop: end
 end subroutine matmul
```

Use --compiler-driven-mode to generate pragmas optimized for the target compiler

codee

Focus the Analysis: Select a subset of checks (V)

```
$ codee checks -- gfortran matmul.f90

CHECKS REPORT

matmul.f90:6:3 [PWR039] (level: L1): Consider loop interchange to improve the locality of reference and enable
                                     vectorization
matmul.f90:1:1 [PWR068] (level: L1): Encapsulate external procedures within modules to avoid the risks of calling
                                     implicit interfaces
matmul.f90 [RMK015] (level: L1): Tune compiler optimization flags to increase the speed of the code
matmul.f90:1:1 [PWR003] (level: L1): Explicitly declare pure functions
matmul.f90:1:1 [PWR008] (level: L1): Declare the intent for each procedure parameter
matmul.f90:1:1 [PWR070] (level: L1): Declare array dummy arguments as assumed-shape arrays
matmul.f90:1:1 [PWR007] (level: L2): Disable implicit declaration of variables
matmul.f90:2:3 [PWR071] (level: L2): Prefer real(kind=kind_value) for declaring consistent floating types
matmul.f90:3:3 [PWR071] (level: L2): Prefer real(kind=kind_value) for declaring consistent floating types
matmul.f90:13:3 [PWR035] (level: L3): Avoid non-consecutive array access to improve performance
matmul.f90:7:5 [RMK010] (level: L3): The vectorization model states the loop is not a SIMD opportunity due to
                                     strided memory accesses
matmul.f90:15:7 [RMK010] (level: L3): The vectorization model states the loop is not a SIMD opportunity due to
                                      strided memory accesses

$ codee checks --check-id PWR007 -- gfortran matmul.f90

CHECKS REPORT

matmul.f90:1:1 [PWR007] (level: L2): Disable implicit declaration of variables
```

Focus the Analysis: Select a subset of code

```
$ codee checks matmul.f90 -- gfortran matmul.f90
```
Filter by file

```
$ codee checks matmul.f90:matmul -- gfortran matmul.f90
```
Filter by function

```
$ codee checks matmul.f90:13 -- gfortran matmul.f90
```
Filter by loop

```
$ codee checks matmul.f90:7,13 -- gfortran matmul.f90
```
Filter by multiple elements

Main Takeaways

- **Simulation software** demands **maintainable** and **high-speed** Fortran/C/C++ code.

- **1. Modernize** your code to ensure **correctness:**
  - Update legacy code; e.g.: F77 → F2018, C++98 → C++20.
  - Ensure portability across compilers; no vendor-specific language extensions.
  - Leverage new language features; e.g.: Fortran modules, C++ smart pointers.
- **2. After that,** address **optimization**.

- **Customize the analysis** to your needs:
  - **Modernization:** `codee --check-id <id> / codee --only-categories modern`
  - **CPU optimization:** `codee --target-arch cpu`
  - **GPU optimization:** `codee --target-arch gpu`
- Look up the **Open Catalog** and leverage **Codee's autofix** capabilities to improve the code:
  - **Modernization autofix:** `codee rewrite --modernization`
  - **(Optimization) CPU + OpenMP autofix:** `codee rewrite --multi omp-for`

Hands-on Demos on Perlmutter @NERSC

- **Live Demo #1:** HIMENO modernization
- **Live Demo #2:** HIMENO optimization through CPU parallelism
- **Live Demo #3:** MATMUL optimization through GPU parallelism

Hands-on Labs on Perlmutter @NERSC

Step-by-step guides available at docs.codee.com:

- PI offloading to GPU at Perlmutter (C/C++)
- MATMUL offloading to GPU at Perlmutter (C/C++)
- COULOMB offloading to GPU at Perlmutter (C/C++)
- HIMENO modernization (Fortran)
- HIMENO optimization through CPU parallelism (Fortran)
- HIMENO optimization through GPU parallelism on NVIDIA/Cray Compilers (Fortran)

codee

Automated Code Inspection for
Modernization and Optimization

www.codee.com

info@codee.com

Subscribe: codee.com/newsletter/

Spain

codee_com

/codee-com/