

## Quickstart guide of Codee

1. Download the binary distribution package of Codee for your system.
2. Uncompress the Codee package in the desired location. This "*Codee installation folder*" contains the subfolder `bin/`, `lib/` and `examples/`.
3. Add the location of the Codee binary available in the subfolder '`bin/`' of the Codee installation folder to the environment PATH. Setup in the system terminal window:

```
Linux:          export PATH="$PATH:<Codee installation folder>/bin"
Windows-terminal: set PATH="%PATH%;<Codee installation folder>\bin"
Windows-powershell: $env:path += "<Codee installation folder>\bin"
MacOS:         export PATH=$PATH:<Codee installation folder>/bin
```

Alternatively, setup the path to the Codee binaries in the computer boot process:

```
Linux-bash:  Add the export PATH command to .bashrc and re-open the terminal
Windows:    Add the export PATH command in System properties and re-open the terminal
```

4. Copy the Codee license file to the *Codee installation folder* with the name "**codee.lic**". Alternatively, specify its location through the `CODEE_LICENSE_PATH` environment variable.
5. Run the Codee command-line tools to show the Codee version installed in the system:  
**codee --version**

6. Using the performance-demos repository as base, go to the `MATMUL/serial` folder:

```
Linux:      git clone https://github.com/codee-com/performance-demos.git
            cd performance-demos/MATMUL/serial
Windows:    git clone https://github.com/codee-com/performance-demos.git
            cd performance-demos\MATMUL\serial
```

7. Produce the Screening Report of Codee (screening command):

```
codee screening main.c -- -I include -O3
```

```
Date: 2024-04-08 Codee version: 2024.2
Compiler flags: -I include -O3
```

```
[C] target compiler: <none> (Compiler Agnostic Mode)
```

```
[1/1] main.c ... Done
```

### SCREENING REPORT

```
---Number of files---
```

Total	C	C++	Fortran
1	1	0	0

Lines of code	Analysis time	# checks	Profiling
55	19 ms	6	n/a

### CHECKS PER CATEGORY AND PRIORITY LEVELS

Checks per category							Priority		
Scalar	Control	Memory	Vector	Multi	Offload	Quality	L1	L2	L3
1	0	3	2	n/a	n/a	0	2	0	4

```
Lines of code : total lines of code found in the target (computed the same way as the sloccount tool)
Analysis time : time required to analyze the target
# checks : total actionable items (opportunities, recommendations, defects and remarks) detected
Profiling : estimation of overall execution time required by this target
```

### RANKING OF CHECKERS

```

Checker Level Priority # Title
-----
-
PWR039 L1 P27 1 Consider loop interchange to improve the locality of reference and enable vectorization
PWR053 L1 P12 1 Consider applying vectorization to forall loop
PWR010 L3 P4 1 Avoid column-major array access in C/C++
PWR048 L3 P3 1 Replace multiplication/addition combo with an explicit call to fused multiply-add
PWR035 L3 P2 1 Avoid non-consecutive array access to improve performance
RMK010 L3 P0 1 The vectorization cost model states the loop is not a SIMD opportunity due to strided memory
accesses in the loop body

SUGGESTIONS

Use 'checks' to find out details about the detected checks:
codee checks main.c -- -I include -O3

Use --target-arch to focus on the checks most relevant to your hardware type [cpu | gpu | mcu], e.g.:
codee screening --target-arch cpu main.c -- -I include -O3

Consider using Codee with a target compiler in order to filter out optimizations that are already applied by your
compiler. For example, for GCC:
codee screening --target-compiler cc gcc main.c -- -I include -O3

1 file, 2 functions, 6 loops successfully analyzed and 0 non-analyzed files in 21 ms

```

- Follow the suggestions in order to produce the Checkers Report (option `--checks`) that lists all the checks applicable to your code.

```
codee checks main.c -- -I include -O3
```

```

Compiler flags: -I include

[C] target compiler: <none> (Compiler Agnostic Mode)

[1/1] main.c ... Done

CHECKS REPORT

main.c:16:9 [PWR039] (level: L1): Consider loop interchange to improve the locality of reference and enable
vectorization
main.c:9:9 [PWR053] (level: L1): Consider applying vectorization to forall loop
main.c:17:13 [PWR010] (level: L3): Avoid column-major array access in C/C++
main.c:18:17 [PWR048] (level: L3): Replace multiplication/addition combo with an explicit call to fused multiply-add
main.c:15:5 [PWR035] (level: L3): Avoid non-consecutive array access to improve performance
main.c:17:13 [RMK010] (level: L3): The vectorization cost model states the loop is not a SIMD opportunity due to
strided memory accesses in the loop body

SUGGESTIONS

Use --verbose to get more details, e.g:
codee checks --verbose main.c -- -I include -O3

Use --level to filter checks with a specific level of priority, e.g:
codee checks --level L1 main.c -- -I include -O3

More details on the defects, recommendations and more in the Open Catalog of Best Practices for Performance:
https://github.com/codee-com/open-catalog/

Consider using Codee with a target compiler in order to filter out optimizations that are already applied by your
compiler. For example, for GCC:
codee checks --target-compiler cc gcc main.c -- -I include -O3

1 file, 2 functions, 6 loops successfully analyzed and 0 non-analyzed files in 20 ms

```

- Show the detailed Checkers Report (option `--verbose`). As an example, focus on the checker *PWR039* related to enforcing memory efficiency on microprocessors through the *Loop Interchange* optimization. The detailed Codee output, which includes links to the open catalog available in the website, precise location in the source code, etc..., is as follows:

```
codee checks main.c --verbose -- -I include -O3
```

```

main.c:16:9 [PWR039] (level: L1): Consider loop interchange to improve the locality of reference and enable
vectorization
  Loops to interchange:
    16:         for (size_t j = 0; j < n; j++) {
    17:             for (size_t k = 0; k < p; k++) {
  Suggestion: Interchange inner and outer loops in the loop nest to improve performance

```

Documentation: <https://github.com/codee-com/open-catalog/tree/main/Checks/PWR039>  
AutoFix:  
codee rewrite --memory loop-interchange --in-place main.c:16:9 -- -I include -O3

10. Note that the check PWR039 contains an *AutoFix* section that remarks the existence of source code rewriting capabilities that facilitate the usage of Codee as a coding assistant. Focus on PWR039 and implement loop interchange in MATMUL by executing the following command:

```
codee rewrite --memory loop-interchange -o main_codee.c main.c:16:9 -- -I include -O3
```

11. Compile the original source code of MATMUL (`main.c`) and the optimized source code of MATMUL (`main_codee.c`). For instance, using the GCC compiler for C language:

```
gcc main.c matrix.c clock.c -o matmul -I include -O3  
gcc main_codee.c matrix.c clock.c -o matmul_codee -O3 -I include
```

12. Run the original MATMUL (`matmul`) and the optimized MATMUL (`matmul_codee`):

Linux:            `./matmul 1500`  
                 `./matmul_codee 1500`

Windows:        `.\matmul 1500`  
                 `.\matmul_codee 1500`

13. Measure the performance improvement obtained in your system. For reference purposes, take a look at the [Codee leaflet for loop interchange](#) which shows up to 3x faster speed on x86 and arm systems equipped with GCC, Clang and Intel compilers.