Tutorial to get started with Codee

Automated Code Inspection for Performance

Shift Left Performance

codee

Revision April 24, 2023

Codee Static Code Analyzer

Codee implements the first static code analysis solution specifically designed to automate performance optimization for C/C++/Fortran applications

Quick Assessment via AI Analysis of Source Code

Pinpoint Opportunities to Optimize Performance: Speed, Size & Energy

Better Testing & Quality Assurance through Performance Sanity Checks

Comprehensive and Accurate Reporting

Save Development Costs and Empower Developers to Deliver Performant Code

Benefits

codee

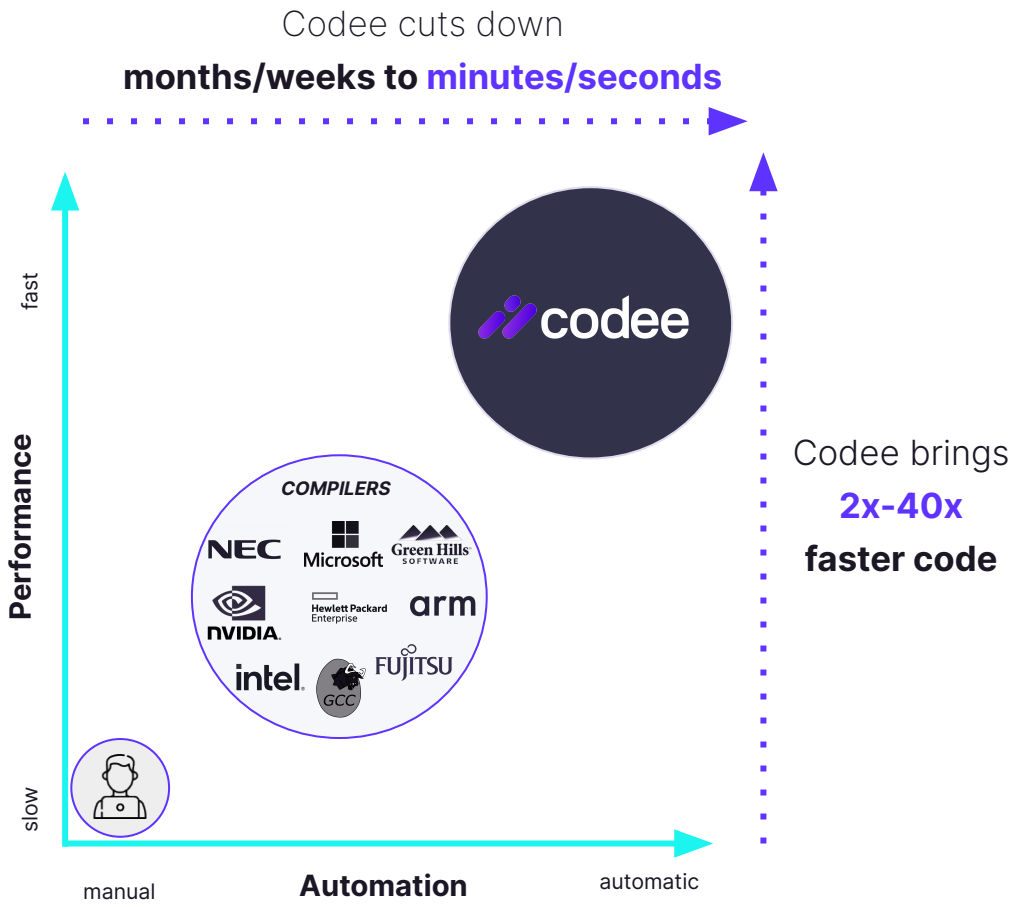
Shift Left Performance

Cost:	Extends hardware lifespan and saves development costs
Time:	Decreases development hours shortening time-to-market
Expertise:	Reduces dependability on expert developers
Energy:	Facilitates development of greener applications

Codee & Ecosystem

Alternatives:

- Do nothing
- Rely on the compiler
- Manual process by experts in performance
- Manual process by external consultancy services



Proof points of Codee | Verifiable & reproducible performance gain

codee

Benchmarking the impact of Codee on the performance achievable in Perlmutter at NERSC

About Perlmutter

Perlmutter, an HPE Cray EX supercomputer delivered to the National Energy Research Scientific Computing Center, is a heterogeneous system with both GPU-accelerated and CPU-only nodes.

The HPE Cray system integrates NVIDIA A100 GPUs, AMD "Milan" EPYC CPUs, a HPE Slingshot high-speed network, and a 35 petabyte – all FLASH – scratch file system.

Perlmutter is designed to meet the emerging simulation, data analytics, and artificial intelligence requirements of the scientific community.

https://www.nersc.gov/systems/perlmutter/
https://perlmutter.carrd.co/

About Codee

Codee is a software development platform that provides automated code inspection specifically designed to improve the performance of C/C++/Fortran applications. It provides a systematic predictable approach to optimize C/C++/Fortran source code for the target environment.

Codee discovers performance optimization opportunities in C/C++/Fortran source code, enabling to benefit from the memory efficiency, vectorization, multithreading and offloading capabilities available in modern computers.

Reproduce the impact on the performance of selected codes representative of embedded and high-performance computing benchmarks. Follow the instructions in the following GitHub repository:

https://github.com/teamappentra/performance-demos/

Benchmarking on Perlmutter using GCC 11.2 (*)

```
$ CC=gcc RUNS=10 RUNS_WARMUP=5
./benchmark-mbedTLS-vector.sh
...
```

Algorithm	Original	Optimized	Speedup
AES-XTS-128	487183 KiB/s	662658 KiB/s	36.02%
AES-XTS-256	437422 KiB/s	554505 KiB/s	26.77%
AES-CBC-128	601075 KiB/s	791688 KiB/s	31.71%
AES-CBC-192	551989 KiB/s	711305 KiB/s	28.86%
AES-CBC-256	510423 KiB/s	644649 KiB/s	26.30%
AES-CMAC-128	572083 KiB/s	777086 KiB/s	35.83%
AES-CMAC-192	561390 KiB/s	698712 KiB/s	24.46%
AES-CMAC-256	514169 KiB/s	631578 KiB/s	22.83%
AES-CMAC-PRF-128	571208 KiB/s	772162 KiB/s	35.18%
ARIA-CBC-128	134467 KiB/s	138524 KiB/s	3.02%
ARIA-CBC-192	118090 KiB/s	121049 KiB/s	2.51%
ARIA-CBC-256	104817 KiB/s	106999 KiB/s	2.08%

```
$ CC=gcc RUNS=10 RUNS_WARMUP=5 ./benchmark-omp-multi.sh
...
```

Code	Original	Optimized	Speedup
ATMUX	0.27	0.14	1.94x
CANNY	5.99	4.62	1.30x
COULOMB	10.23	0.13	77.69x
HACCmk	35.98	4.45	8.08x
MATMUL	3.01	0.06	52.07x
NPB_CG	24.61	5.24	4.70x
PI	3.84	0.05	72.50x

(*) Benchmarking conducted on Perlmutter using Codee 1.6.0-186 (Jan 12, 2023) on a remote machine equipped with AMD EPYC 7763 64-Core CPU and SUSE Linux Enterprise Server 15 SP4 Operating System and GCC 11.2 compiler. Performance measured by running 5 warm-up runs and calculating the average of 10 runs, using Slurm (salloc --nodes 1 --qos interactive --time 01:00:00 --constraint cpu --account=nguest). The compiler optimization levels are as follows: MBedTLS release with -O2 and MULTI benchmarks with -O3.

About NERSC

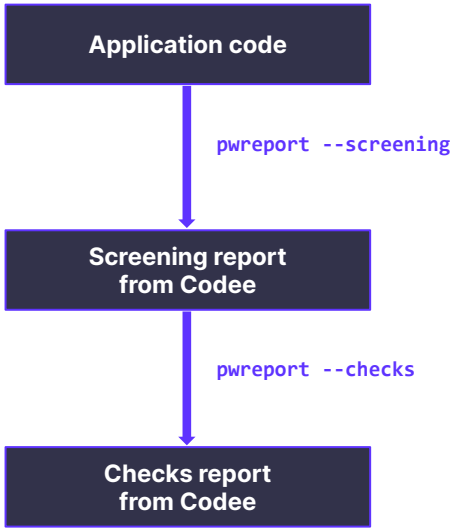
The National Energy Research Scientific Computing Center (NERSC) is a U.S. Department of Energy Office of Science User Facility that serves as the primary high-performance computing center for scientific research sponsored by the Office of Science. Located at Lawrence Berkeley National Laboratory, the NERSC Center serves more than 8,000 scientists at national laboratories and universities researching a wide range of problems in combustion, climate modeling, fusion energy, materials science, physics, chemistry, computational biology, and other disciplines.

NERSC

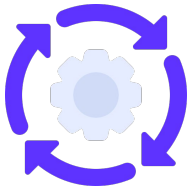
www.codee.com | info@codee.com | +34 881015556

March 6, 2023

How does Codee Static Code Analyzer work?



- **Quick assessment**
- **Checks report**
- **Support for target environments** (operating system, compiler, hardware)
- **Integration with DevOps CI/CD**



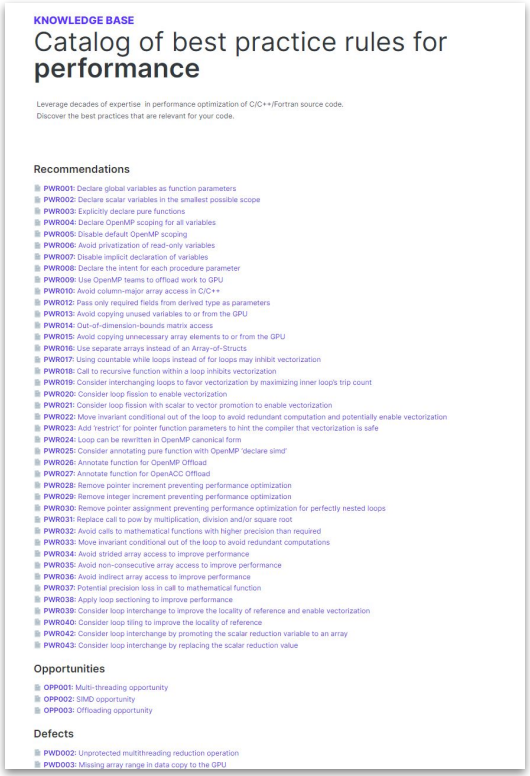
First, Produce the Codee Screening Report

```
$ pwreport --screening --config build/compile_commands.json --show-progress
. . .
SCREENING REPORT

Target                      Lines of code Optimizable lines Analysis time # checks  Effort  Cost      Profiling
--------------------------- ------------- ----------------- ------------- --------- ------- --------- ---------
build/compile_commands.json 325804        28095             1 m 39 s      310       1765 h  57758€    n/a
--------------------------- ------------- ----------------- ------------- --------- ------- --------- ---------
Total                       325804        28095             1 m 39 s      310       1765 h  57758€    n/a

. . .

TOTAL NUMBER OF LOOPS FOR CODEE AUTO AND GUIDED MODES
                    --------------- # checks ---------------
Codee mode # Loops Scalar Control Memory Vector Multi Offload
---------- ------- ------ ------- ------ ------ ----- -------
Auto       105     0      0       0      105    n/a   n/a
Guided     136     17     0       137    45     n/a   n/a

Codee mode : Available Codee mode for the loop:
  - Auto: Codee optimizes the code automatically
  - Guided: Codee identifies the performance issue, and the programmer must apply the changes to the code

SUGGESTIONS

  Use --checks to find out details about the detected checks:
        pwreport --checks --config build/compile_commands.json --show-progress


370 files successfully analyzed and 0 failures in 1 m 39 s
```

Second, Produce the Codee Checks Report

```
$ pwreport --checks --verbose library/aes.c:mbedtls_aes_crypt_xts --config build/compile_commands.json
. . .
CHECKS REPORT

library/aes.c:1162:9 [PWR053]: consider applying vectorization to forall loop
  Suggestion: use pwdirectives to automatically optimize the code
  Documentation: https://www.codee.com/knowledge/pwr053
  AutoFix (choose one option):
        * Using OpenMP pragmas (recommended):
        pwdirectives --vector omp --in-place library/aes.c:1162:9 --config build/compile_commands.json
        * Using Clang compiler pragmas:
        pwdirectives --vector clang --in-place library/aes.c:1162:9 --config build/compile_commands.json
        * Using GCC pragmas
        pwdirectives --vector gcc --in-place library/aes.c:1162:9 --config build/compile_commands.json
        * Using ICC pragmas:
        pwdirectives --vector icc --in-place library/aes.c:1162:9 --config build/compile_commands.json

. . .


library/aes.c:1202:9 [PWR024]: the loop is currently not in OpenMP canonical form
  Suggestion: rewrite the loop in order to comply with the OpenMP Canonical Loop Form
  Documentation: https://www.codee.com/knowledge/pwr024

. . .

1 file successfully analyzed and 0 failures in 597 ms
```

Open Catalog of Performance Optimization Best Practices



KNOWLEDGE BASE

Catalog of best practice rules for **performance**

Leverage decades of expertise in performance optimization of C/C++/Fortran source code.
Discover the best practices that are relevant for your code.

Recommendations

- **PWR001**: Declare global variables as function parameters
- **PWR002**: Declare scalar variables in the smallest possible scope
- **PWR003**: Explicitly declare pure functions
- **PWR004**: Declare OpenMP scoping for all variables
- **PWR005**: Disable default OpenMP scoping
- **PWR006**: Avoid privatization of read-only variables
- **PWR007**: Disable implicit declaration of variables
- **PWR008**: Declare the intent for each procedure parameter
- **PWR009**: Use OpenMP teams to offload work to GPU
- **PWR010**: Avoid column-major array access in C/C++
- **PWR012**: Pass only required fields from derived type as parameters
- **PWR013**: Avoid copying unused variables to or from the GPU
- **PWR014**: Out-of-dimension-bounds matrix access
- **PWR015**: Avoid copying unnecessary array elements to or from the GPU
- **PWR016**: Use separate arrays instead of an Array-of-Structs
- **PWR017**: Using countable while loops instead of for loops may inhibit vectorization
- **PWR018**: Call to recursive function within a loop inhibits vectorization
- **PWR019**: Consider interchanging loops to favor vectorization by maximizing inner loop's trip count
- **PWR020**: Consider loop fission to enable vectorization
- **PWR021**: Consider loop fission with scalar to vector promotion to enable vectorization
- **PWR022**: Move invariant conditional out of the loop to avoid redundant computation and potentially enable vectorization
- **PWR023**: Add 'restrict' for pointer function parameters to hint the compiler that vectorization is safe
- **PWR024**: Loop can be rewritten in OpenMP canonical form
- **PWR025**: Consider annotating pure function with OpenMP 'declare simd'
- **PWR026**: Annotate function for OpenMP Offload
- **PWR027**: Annotate function for OpenACC Offload
- **PWR028**: Remove pointer increment preventing performance optimization
- **PWR029**: Remove integer increment preventing performance optimization
- **PWR030**: Remove pointer assignment preventing performance optimization for perfectly nested loops
- **PWR031**: Replace call to pow by multiplication, division and/or square root
- **PWR032**: Avoid calls to mathematical functions with higher precision than required
- **PWR033**: Move invariant conditional out of the loop to avoid redundant computations
- **PWR034**: Avoid strided array access to improve performance
- **PWR035**: Avoid non-consecutive array access to improve performance
- **PWR036**: Avoid indirect array access to improve performance
- **PWR037**: Potential precision loss in call to mathematical function
- **PWR038**: Apply loop sectioning to improve performance
- **PWR039**: Consider loop interchange to improve the locality of reference and enable vectorization
- **PWR040**: Consider loop tiling to improve the locality of reference
- **PWR042**: Consider loop interchange by promoting the scalar reduction variable to an array
- **PWR043**: Consider loop interchange by replacing the scalar reduction value

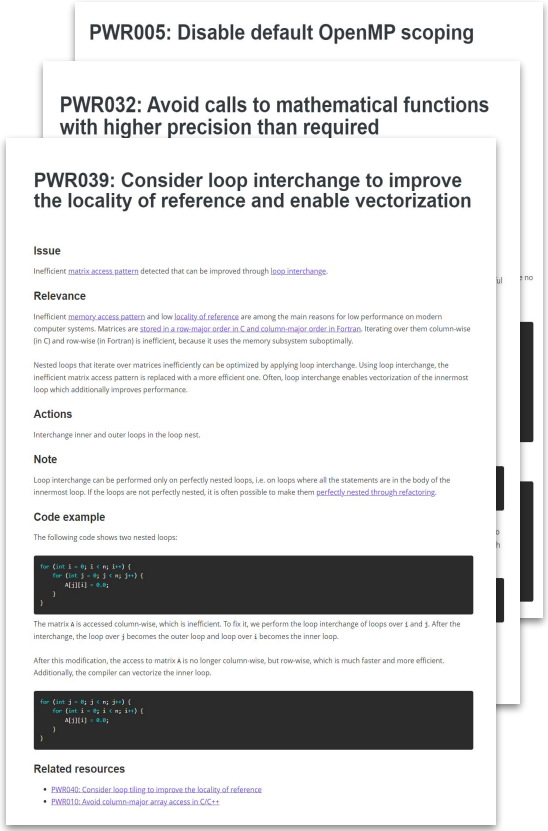
Opportunities

- **OPP001**: Multi-threading opportunity
- **OPP002**: SIMD opportunity
- **OPP003**: Offloading opportunity

Defects

- **PWD002**: Unprotected multithreading reduction operation
- **PWD003**: Missing array range in data copy to the GPU

https://www.codee.com/knowledge/



PWR005: Disable default OpenMP scoping

PWR032: Avoid calls to mathematical functions with higher precision than required

PWR039: Consider loop interchange to improve the locality of reference and enable vectorization

Issue

Inefficient matrix access pattern detected that can be improved through loop interchange.

Relevance

Inefficient memory access pattern and low locality of reference are among the main reasons for low performance on modern computer systems. Matrices are stored in a row-major order in C and column-major order in Fortran. Iterating over them column-wise (in C) and row-wise (in Fortran) is inefficient, because it uses the memory subsystem suboptimally.

Nested loops that iterate over matrices inefficiently can be optimized by applying loop interchange. Using loop interchange, the inefficient matrix access pattern is replaced with a more efficient one. Often, loop interchange enables vectorization of the innermost loop which additionally improves performance.

Actions

Interchange inner and outer loops in the loop nest.

Note

Loop interchange can be performed only on perfectly nested loops, i.e. on loops where all the statements are in the body of the innermost loop. If the loops are not perfectly nested, it is often possible to make them perfectly nested through refactoring.

Code example

The following code shows two nested loops:

```
for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
      A[i][i] = 0.0;
   }
}
```

The matrix `A` is accessed column-wise, which is inefficient. To fix it, we perform the loop interchange of loops over `i` and `j`. After the interchange, the loop over `j` becomes the outer loop and loop over `i` becomes the inner loop.

After this modification, the access to matrix `A` is no longer column-wise, but row-wise, which is much faster and more efficient. Additionally, the compiler can vectorize the inner loop.

```
for (int j = 0; j < n; j++) {
   for (int i = 0; i < n; i++) {
      A[i][i] = 0.0;
   }
}
```

Related resources

- PWR040: Consider loop tiling to improve the locality of reference
- PWR010: Avoid column-major array access in C/C++

Codee goes beyond the State of the Art: Loop Interchange

codee

Performance - Portability across compilers through Loop Interchange

Why is Loop Interchange important?

Loop interchange is a performance optimization technique that is used to improve the loop's memory access pattern and potentially enable vectorization. Loop interchange if applied correctly can yield a huge performance improvement.

Loop interchange as an optimization technique is applicable to loop nests. A loop nest consists of two or more nested loops. After loop interchange, a loop that is once outer has now become inner, and the loop that was once inner has become outer.

Automation of Loop Interchange

Loop interchange is not automated in current tools, so Codee brings new innovation in this space. Codee has built-in support to detect loop nests where Loop Interchange can bring a huge performance gain through PWRs (PWR039, PWR042, PWR043), and it is work-in-progress to fully automate source code rewriting. Consider the implementation of the matrix-matrix multiplication shown below:

```
28  void matmul(int n, double *A, double *B, double *C) {
30    for (int i = 0; i < n; ++i)
31    {
32      for (int j = 0; j < n; ++j)
33      {
34        double c = 0.0;
35        for (int k = 0; k < n; ++k)
36        {
37          c += A[i * n + k] * B[k * n + j];
38        }
39        C[i * n + j] += c;
40      }
41    }
42  }
```

Running Codee for this code, the PWR042 is found and reported to the user. After rewriting the source code according to best practice, the following loop is created:

```
28  void matmul(int n, double *A, double *B, double *C) {
30    for (int i = 0; i < n; ++i)
31    {
32      for (int k = 0; k < n; ++k)
33      {
34        for (int j = 0; j < n; ++j)
35        {
36          C[i * n +j] += A[i * n + k] * B[k * n + j];
37        }
38      }
39    }
40  }
```

Note that after applying loop interchange, the ordering of the nested loops changes from IJK to IKJ. Also it favours sequential memory accesses in the innermost loop, which enables its vectorization.

Performance Evaluation

The performance improvement achieved through loop interchange in the example matrix-matrix multiplication code is evaluated on x86 and Arm processors, resulting in 2x-3x faster code.

Environment Linux Arm	Before Codee (seconds)	After Codee (seconds)	Speedup
CLANG 14 -O3 -ffast-math	353.74	181.34	48.73% **(1.95x)**
GCC 11 -O3 -ffast-math	329.59	188.97	42.66% **(1.74x)**
armclang -O3 -ffast-math	357.39	181.73	49.15% **(1.97x)**

Codee brings **2x faster** code on Arm environments through loop interchange and vectorization

Environment Linux x86_64	Before Codee (seconds)	After Codee (seconds)	Speedup
CLANG 14 -O3 -ffast-math	9.485	3.333	64.86% **(2.85x)**
GCC 11 -O3 -ffast-math	8.757	3.457	64.86% **(2.85x)**
ICC -O3 -ffast-math	7.342	3.261	64.86% **(2.85x)**

Codee brings **3x faster** code on x86 environments through loop interchange and vectorization

From the technology perspective, Codee automates loop interchange and enables the efficient vectorization of the innermost loop. Note that GNU, LLVM and Intel compilers do not apply loop interchange in the maximum performance optimization level setup. In practice, this is a demonstrator of Codee advancing the state-of-the-art of compilers.

Innermost loop matmul_naive:35	GCC 11 -O3 -ffast-math	CLANG 14 -O3 -ffast-math	armclang -O3 -ffast-math	ICC -O3 -ffast-math	Codee
Loop interchange					YES
Loop vectorization	YES	NO (cost model)	NO (cost model)	YES	YES
Loop peeling				YES	
Loop interleaving		NO (cost model)	NO (cost model)		
Loop turned into non-loop	YES				

Download link: https://www.codee.com/wp-content/uploads/2023/01/Leaflet-Loop-Interchange.pdf

Codee Static Code Analyzer (& Coding Assistant)

Codee also implements a coding assistant to help develop performance-portable C/C++/Fortran code for GPU-based supercomputers using the industry standard compiler directives

Code rewriting features for C/C++/Fortran source code: guided & auto modes

Annotation of source code for offloading with compiler directives: OpenMP & OpenACC

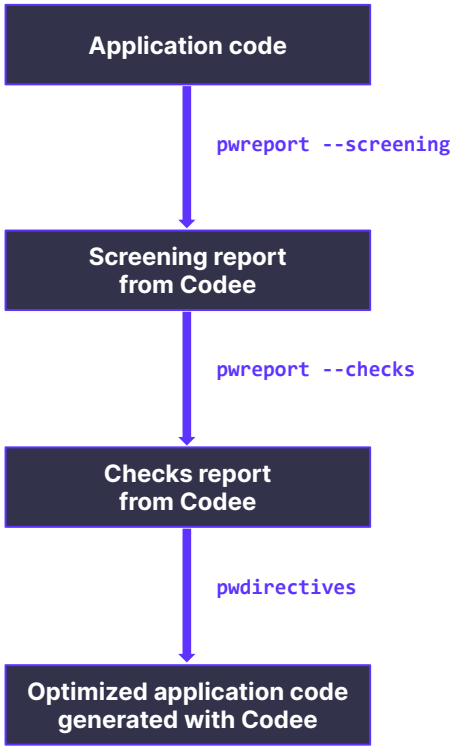
Annotation of source code for single-core optimization: ISA extensions, memory, control & vectorization

Annotation of source code for multi-core optimization: OpenMP multithreading

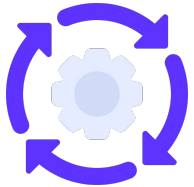
Analysis of the performance optimizations reported by compilers: GCC, LLVM, ICC, CL

Analysis of memory usage: memory access patterns, data scoping, memory footprint

How to enable Codee's coding assistant capabilities?



- **Quick assessment**
- **Checks report**
- **Support for target environments** (operating system, compiler, hardware)
- **Integration with DevOps CI/CD**
- **New optimized code**



How to enable Codee's coding assistant capabilities?

```
$ pwreport --screening --config build/compile_commands.json --verbose library/aes.c
SCREENING REPORT

Target        Lines of code Optimizable lines Analysis time # checks Effort Cost    Profiling
------------- ------------- ----------------- ------------- -------- ------ ------- ---------
library/aes.c 1657          246               481 ms        20       111 h  3632€   n/a
------------- ------------- ----------------- ------------- -------- ------ ------- ---------
Total         1657          246               481 ms        20       111 h  3632€   n/a

. . .

TOTAL NUMBER OF LOOPS FOR CODEE AUTO AND GUIDED MODES
                                                 ---------------- # checks ----------------
Codee mode File :function              # Loops Scalar Control Memory Vector Multi Offload
---------- --------------------------- ------- ------ ------- ------ ------ ----- -------
Auto                                   7       0      0       0      7      n/a   n/a
           library/aes.c
           |- :mbedtls_aes_crypt_cbc   2       0      0       0      2      n/a   n/a
           `- :mbedtls_aes_crypt_xts   5       0      0       0      5      n/a   n/a
---------- --------------------------- ------- ------ ------- ------ ------ ----- -------

Codee mode : Available Codee mode for the loop:
  - Auto: Codee optimizes the code automatically
  - Guided: Codee identifies the performance issue, and the programmer must apply the changes to the code

SUGGESTIONS

  You can automatically vectorize every vectorizable loop of one function with:
        pwdirectives --auto --simd omp --in-place --config build/compile_commands.json library/aes.c


1 file successfully analyzed and 0 failures in 481 ms
```

Third, automatically rewrite & verify the optimized code

```
$ pwdirectives --auto --simd omp --in-place --config build/compile_commands.json library/aes.c:mbedtls_aes_crypt_xts

C target compiler: /usr/bin/gcc, version 9.4.0

Loop                                GCC-9.4.0  Codee 1.6.0  Verified by Codee
----------------------------------- ---------- ------------ ------------------------
library/aes.c
|- mbedtls_aes_crypt_xts:1126:5     n/a        no (outr)
|- mbedtls_aes_crypt_xts:1127:5     n/a        no (outr)
|- mbedtls_aes_crypt_xts:1129:5     n/a        no (outr)
|- mbedtls_aes_crypt_xts:1130:5     n/a        no (outr)
|- mbedtls_aes_crypt_xts:1131:5     n/a        no (outr)
|- mbedtls_aes_crypt_xts:1147:5     n/a        no (outr)
|  |- mbedtls_aes_crypt_xts:1162:9 n/a        auto         vectorized by GCC-9.4.0
|  `- mbedtls_aes_crypt_xts:1169:9 n/a        auto         vectorized by GCC-9.4.0
|- mbedtls_aes_crypt_xts:1194:9     no (othr)  auto         vectorized by GCC-9.4.0
|- mbedtls_aes_crypt_xts:1202:9     no (othr)  yes
`- mbedtls_aes_crypt_xts:1211:9     auto       skipped

Loop : loop name following the syntax <file>:<function>:<line>:<column>

Codee 1.6.0 - Codee vectorization status:
    skipped: the target compiler vectorized the loop automatically, so that Codee skipped it
    auto: Codee is able to automatically rewrite the loop using SIMD pragmas
    yes: Codee detect it as a SIMD opportunity, but it is unable to rewrite it automatically
    no: Codee vectorization cost model determines that the loop is not a worthwhile SIMD opportunity. The reason is
indicated in brackets (same options as compiler column)

Verified by Codee - Explicit vectorization profitable ('vectorized by GCC-9.4.0' when Codee successfully produced an
explicit omp|gcc|clang SIMD pragma that resulted in automatic compiler vectorization, 'not vectorized by GCC-9.4.0' for
all the other cases)

1 file successfully analyzed and 0 failures in 656 ms
```

Challenges in the development of GPU-enabled code

Challenges for GPU acceleration	Find opportunities for offloading	Optimize memory layout for data transfers	Identify defects in data transfers	Exploit massive parallelism through loop nest collapsing	Minimize data transfers across consecutive loop nests	Minimize data transfers through convergence loops	Identify auxiliary functions to be offloaded
MATMUL	X	X	X	X	X		
PI	X						
LULESHmk	X	X				X	X
MATMUL PWD006			X				
ATMUX	X						
ZPIC	X	X	X		X	X	
MBEDTLS	**Opportunities for vectorization**	-	-	-	-	-	-
NUCCOR	X						
Your code!	Probably all of these challenges apply, and even more!						

Demo #1: Codee Static Code Analyzer & Coding Assistant

- **Codee CLI:** *pwreport --screening*
  pwreport --checks
  pwreport --checks --verbose
  pwdirectives
- **Environment**: Perlmutter with Nvidia/GNU compilers
- **Application**: MATMUL-C (matrix-matrix multiplication)
- **Takeaways**:
  - Codee found 8 checks applicable to MATMUL, out of 50+ checks available in the open catalog.
  - Codee reports checks to help enforcing performance optimization best practices on the code, related to memory efficiency, offloading, multi-threading and vectorization.
  - First, enforce single-core optimations through Loop Interchange (PWR039).
  - Second, enable offloading to GPU using OpenACC and OpenMP.
  - Benchmarking on Perlmutter: Reduce runtime from 39.6 seconds down to 5-6 seconds.
  - Overall speedup on Perlmutter GPUs is 6x-8x (using OpenACC and OpenMP).
  - Codee helps creating performance-portable code across compilers.

Automated Code Inspection for Performance

www.codee.com

info@codee.com

Subscribe: codee.com/newsletter/

Spain

codee_com

/codee-com/