codee

Quickstart Guide of Codee MBedTLS Cryptographic Library (MBEDTLS) Step-by-Step in Perlmutter @NERSC

Step 1. Setup the environment

Load Codee last version and the GNU programming environment [1] through the modules management system:

```
$ module load codee
$ module load PrgEnv-gnu
```

Run Codee configuration wizard to set up the compiler used to generate executables. In this case, select the option "*gcc-11.2 (/opt/cray/pe/gcc/11.2.0/bin/gcc*" [2], meaning that Codee will attempt to interpret the user messages emitted by the compiler.

```
$ pwreport --configuration-wizard
```

Step 2: Understanding the implementation of MBEDTLS

MbedTLS is an open source C library that implements cryptographic primitives, X.509 certificate manipulation and the TLS and DTLS protocols. Its small code footprint makes it suitable for embedded systems. It is a solution endorsed by ARM, and a very common choice for implementing TLS in embedded systems. As an example, the source code below shows the implementation of the AES encryption algorithm, in particular a snippet of the source file "*library/aes.c*".

```
1109    /*
1110     * AES-XTS buffer encryption/decryption
1111     */
1112    int mbedtls_aes_crypt_xts( mbedtls_aes_xts_context *ctx,
1113                               int mode,
1114                               size_t length,
```

[1] Note we will use the Nvidia compiler *nvc* to generate executables for GPUs.
[2] Note Codee 2023.04 supports the GNU, LLVM, Intel and Microsoft compilers. The Nvidia compiler is not supported.

```
1115                                const unsigned char data_unit[16],
1116                                const unsigned char *input,
1117                                unsigned char *output )
1118    {
1119            int ret = MBEDTLS_ERR_ERROR_CORRUPTION_DETECTED;
1120            size_t blocks = length / 16;
1121            size_t leftover = length % 16;
1122            unsigned char tweak[16];
1123            unsigned char prev_tweak[16];
1124            unsigned char tmp[16];
1125
1126            AES_VALIDATE_RET( ctx != NULL );
1127            AES_VALIDATE_RET( mode == MBEDTLS_AES_ENCRYPT ||
1128                    mode == MBEDTLS_AES_DECRYPT );
1129            AES_VALIDATE_RET( data_unit != NULL );
1130            AES_VALIDATE_RET( input != NULL );
1131            AES_VALIDATE_RET( output != NULL );
1132
1133            /* Data units must be at least 16 bytes long. */
1134            if( length < 16 )
1135            return MBEDTLS_ERR_AES_INVALID_INPUT_LENGTH;
1136
1137            /* NIST SP 800-38E disallows data units larger than 2**20 blocks. */
1138            if( length > ( 1 << 20 ) * 16 )
1139            return MBEDTLS_ERR_AES_INVALID_INPUT_LENGTH;
1140
1141            /* Compute the tweak. */
1142            ret = mbedtls_aes_crypt_ecb( &ctx->tweak, MBEDTLS_AES_ENCRYPT,
1143                                         data_unit, tweak );
1144            if( ret != 0 )
1145            return( ret );
1146
1147            while( blocks-- )
1148            {
1149            size_t i;
1150
1151            if( leftover && ( mode == MBEDTLS_AES_DECRYPT ) && blocks == 0 )
1152            {
1153            /* We are on the last block in a decrypt operation that has
1154                     * leftover bytes, so we need to use the next tweak for this block,
1155                     * and this tweak for the lefover bytes. Save the current tweak for
1156                     * the leftovers and then update the current tweak for use on this,
1157                     * the last full block. */
1158            memcpy( prev_tweak, tweak, sizeof( tweak ) );
1159            mbedtls_gf128mul_x_ble( tweak, tweak );
1160            }
1161
1162            for( i = 0; i < 16; i++ )
1163            tmp[i] = input[i] ^ tweak[i];
1164
1165            ret = mbedtls_aes_crypt_ecb( &ctx->crypt, mode, tmp, tmp );
1166            if( ret != 0 )
1167            return( ret );
1168
1169            for( i = 0; i < 16; i++ )
1170            output[i] = tmp[i] ^ tweak[i];
1171
1172            /* Update the tweak for the next block. */
1173            mbedtls_gf128mul_x_ble( tweak, tweak );
1174
1175            output += 16;
1176            input += 16;
1177            }
1178
1179            if( leftover )
1180            {
1181            /* If we are on the leftover bytes in a decrypt operation, we need to
1182            * use the previous tweak for these bytes (as saved in prev_tweak). */
1183            unsigned char *t = mode == MBEDTLS_AES_DECRYPT ? prev_tweak : tweak;
1184
1185            /* We are now on the final part of the data unit, which doesn't divide
1186            * evenly by 16. It's time for ciphertext stealing. */
1187            size_t i;
1188            unsigned char *prev_output = output - 16;
1189
1190            /* Copy ciphertext bytes from the previous block to our output for each
1191            * byte of cyphertext we won't steal. At the same time, copy the
```

```
1192            * remainder of the input for this final round (since the loop bounds
1193            * are the same). */
1194            for( i = 0; i < leftover; i++ )
1195            {
1196            output[i] = prev_output[i];
1197            tmp[i] = input[i] ^ t[i];
1198            }
1199
1200            /* Copy ciphertext bytes from the previous block for input in this
1201            * round. */
1202            for( ; i < 16; i++ )
1203            tmp[i] = prev_output[i] ^ t[i];
1204
1205            ret = mbedtls_aes_crypt_ecb( &ctx->crypt, mode, tmp, tmp );
1206            if( ret != 0 )
1207            return ret;
1208
1209            /* Write the result back to the previous block, overriding the previous
1210            * output we copied. */
1211            for( i = 0; i < 16; i++ )
1212            prev_output[i] = tmp[i] ^ t[i];
1213            }
1214
1215            return( 0 );
1216    }
1217    #endif /* MBEDTLS_CIPHER_MODE_XTS */
```

Step 3: Producing the Codee Screening Report

The Codee Screening Report is intended to help assess the speed-up potential of the code. Invoke the *pwreport --screening* command. The default set up enables single-core optimizations on the CPU (note multithreading and GPU offload are disabled by default). The Codee best practice recommendation is to start with the analysis of the open-source project as a whole. For this purpose, let's take advantage of Codee integration with build systems through the compile commands JSON file. The commands below show how to produce it from *CMake*, and how to pass the JSON file to the *pwreport* tool. Note the screening shows the gcc compiler optimization flag (*-O2*) and the progress through all the files of the project (*--show-progress*).

```
$ cmake -DENABLE_TESTING=On -DCMAKE_C_COMPILER=gcc -DUSE_SHARED_MBEDTLS_LIBRARY=On -DCMAKE_BUILD_TYPE=Release
-DCMAKE_EXPORT_COMPILE_COMMANDS=1 -DMBEDTLS_FATAL_WARNINGS=Off -B build -G "Unix Makefiles"

$ make -C build/

$ pwreport --screening --config build/compile_commands.json --show-progress
pwreport: warning: repeated files found in the provided compilation database. Ignoring additional occurrences.
[C] target compiler '/opt/cray/pe/gcc/11.2.0/bin/gcc', 11.2.0
  Full version name: gcc version 11.2.0 20210728 (Cray Inc.) (GCC)
  Optimization flags: -O2

[  1/263] /global/homes/s/user/MbedTLS/tests/src/asn1_helpers.c
[  2/263] /global/homes/s/user/MbedTLS/tests/src/certs.c
[  3/263] /global/homes/s/user/MbedTLS/tests/src/drivers/hash.c
[  4/263] /global/homes/s/user/MbedTLS/tests/src/drivers/platform_builtin_keys.c
[  5/263] /global/homes/s/user/MbedTLS/tests/src/drivers/test_driver_aead.c
[  6/263] /global/homes/s/user/MbedTLS/tests/src/drivers/test_driver_cipher.c
[  7/263] /global/homes/s/user/MbedTLS/tests/src/drivers/test_driver_key_management.c
[  8/263] /global/homes/s/user/MbedTLS/tests/src/drivers/test_driver_mac.c
[  9/263] /global/homes/s/user/MbedTLS/tests/src/drivers/test_driver_signature.c
[ 10/263] /global/homes/s/user/MbedTLS/tests/src/fake_external_rng_for_test.c
[ 11/263] /global/homes/s/user/MbedTLS/tests/src/helpers.c
[ 12/263] /global/homes/s/user/MbedTLS/tests/src/psa_crypto_helpers.c
[ 13/263] /global/homes/s/user/MbedTLS/tests/src/psa_exercise_key.c
[ 14/263] /global/homes/s/user/MbedTLS/tests/src/random.c
[ 15/263] /global/homes/s/user/MbedTLS/tests/src/threading_helpers.c
[ 16/263] /global/homes/s/user/MbedTLS/library/aes.c
```

```
...
[ 18/263] /global/homes/s/user/MbedTLS/library/aria.c
...
[ 30/263] /global/homes/s/user/MbedTLS/library/cmac.c
...
[262/263] /global/homes/s/user/MbedTLS/build/tests/test_suite_net.c
[263/263] /global/homes/s/user/MbedTLS/build/tests/test_suite_oid.c

SCREENING REPORT

Target                       Lines of code Optimizable lines Analysis time # checks Effort Cost     Profiling
---------------------------- ------------- ----------------- ------------- -------- ------ -------- ---------
build/compile_commands.json 245177        296341            1 m 11 s      238      1198 h 39203€   n/a
---------------------------- ------------- ----------------- ------------- -------- ------ -------- ---------
Total                        245177        296341            1 m 11 s      238      1198 h 39203€   n/a

CHECKS PER STAGE OF THE PERFORMANCE OPTIMIZATION ROADMAP
Target                       Scalar Control Memory Vector Multi Offload
---------------------------- ------ ------- ------ ------ ----- -------
build/compile_commands.json 14     0       148    74     n/a   n/a
---------------------------- ------ ------- ------ ------ ----- -------
Total                        14     0       148    74     n/a   n/a

TOTAL NUMBER OF LOOPS FOR CODEE AUTO AND GUIDED MODES
                  ---------------- # checks ----------------
Codee mode # Loops Scalar Control Memory Vector Multi Offload
---------- ------- ------ ------- ------ ------ ----- -------
Auto       52      0      0       0      52     n/a   n/a
Guided     102     12     0       148    22     n/a   n/a

Target : analyzed directory or source code file
Lines of code : total lines of code found in the target (computed the same way as the sloccount tool)
Optimizable lines : relevant lines of code that Codee detects as optimizable
Analysis time : time required to analyze the target
# checks : total actionable items (opportunities, recommendations, defects and remarks) detected
Effort : estimated number of hours it would take to carry out all checks (scalar, control, memory, vector, multi and
offload with 1, 2, 4, 8, 12 and 16 hours respectively)
Cost : estimated cost in euros to carry out all the checks, paying the average salary of 56,286€/year for a
professional C/C++ developer working 1720 hours per year
Profiling : estimation of overall execution time required by this target

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 --verbose to get more details, e.g:
        pwreport --verbose --screening --config build/compile_commands.json --show-progress

  You can specify multiple inputs which will be displayed as multiple rows (ie. targets) in the table, eg:
        pwreport --screening some/other/dir --config build/compile_commands.json --show-progress

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

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

  Multithreading and offloading checks are filtered by default. Use --include-tags to enable them:
        pwreport --screening --include-tags all --config build/compile_commands.json --show-progress

  You can focus on a specific optimization type by filtering by its tag (scalar, control, memory, vector, multi,
offload), eg.:
        pwreport --checks --include-tags scalar --config build/compile_commands.json --show-progress

263 files successfully analyzed and 0 failures in 1 m 12 s
```

Typically, the screening points to code snippets that can be optimized using Codee. Codee does not do a profiling of the application, so it is good practice to dig deeper in the analysis of source code functions/loops that consume a large amount of runtime (i.e. *hotspots*). Right below, the screening of the hotspot source file "*library/aes.c*" is shown, enabling the verbose mode (*--verbose*) that identifies the loops of the source code where Codee source code rewriting capabilities are useful (see files/functions listed in *Auto* mode). In the file "*aes.c*", there are a total of 4 loops that can be vectorized using Codee, and that were overlooked by GCC, spread across the functions *mbedtls_aes_crypt_xts* and *mbedtls_aes_crypt_cbc*.

```
$ pwreport --screening --verbose --config build/compile_commands.json library/aes.c
[C] target compiler '/opt/cray/pe/gcc/11.2.0/bin/gcc', 11.2.0
  Full version name: gcc version 11.2.0 20210728 (Cray Inc.) (GCC)
  Optimization flags: -O2

SCREENING REPORT

Target        Lines of code Optimizable lines Analysis time # checks Effort Cost    Profiling
------------- ------------- ----------------- ------------- -------- ------ ------- ---------
library/aes.c 1657          44675             1474 ms       22       119 h  3894€   n/a
------------- ------------- ----------------- ------------- -------- ------ ------- ---------
Total         1657          44675             1474 ms       22       119 h  3894€   n/a

CHECKS PER STAGE OF THE PERFORMANCE OPTIMIZATION ROADMAP
Target        Scalar Control Memory Vector Multi Offload
------------- ------ ------- ------ ------ ----- -------
library/aes.c 3      0       7      11     n/a   n/a
------------- ------ ------- ------ ------ ----- -------
Total         3      0       7      11     n/a   n/a

LIST OF FUNCTIONS 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
           /global/homes/s/user/performance-demos/MbedTLS/v3.1.0/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
---------- ------------------------------- ------- ------ ------- ------ ------ ----- -------
Guided                                       8       3      0       7      4      n/a   n/a
           /global/homes/s/user/performance-demos/MbedTLS/v3.1.0/library/aes.c
           |- :aes_gen_tables               2       0      0       2      3      n/a   n/a
           |- :mbedtls_aes_setkey_enc       4       3      0       3      1      n/a   n/a
           |- :mbedtls_internal_aes_encrypt 1       0      0       1      0      n/a   n/a
           `- :mbedtls_internal_aes_decrypt 1       0      0       1      0      n/a   n/a
---------- ------------------------------- ------- ------ ------- ------ ------ ----- -------

Target : analyzed directory or source code file
Lines of code : total lines of code found in the target (computed the same way as the sloccount tool)
Optimizable lines : relevant lines of code that Codee detects as optimizable
Analysis time : time required to analyze the target
# checks : total actionable items (opportunities, recommendations, defects and remarks) detected
Effort : estimated number of hours it would take to carry out all checks (scalar, control, memory, vector, multi and
offload with 1, 2, 4, 8, 12 and 16 hours respectively)
Cost : estimated cost in euros to carry out all the checks, paying the average salary of 56,286€/year for a
professional C/C++ developer working 1720 hours per year
Profiling : estimation of overall execution time required by this target

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 specify multiple inputs which will be displayed as multiple rows (ie. targets) in the table, eg:
  You can specify multiple inputs which will be displayed as multiple rows (ie. targets) in the table, eg:
        pwreport --screening some/other/dir --verbose --config build/compile_commands.json library/aes.c

  Use --checks to find out details about the detected checks:
        pwreport --checks --verbose --config build/compile_commands.json library/aes.c
```

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

  Use pwloops to see the entire list of loops with further details (e.g. loop mode, compute patterns, variable scope):
        pwloops --config build/compile_commands.json library/aes.c

  Multithreading and offloading checks are filtered by default. Use --include-tags to enable them:
        pwreport --screening --include-tags all --verbose --config build/compile_commands.json library/aes.c

  You can focus on a specific optimization type by filtering by its tag (scalar, control, memory, vector, multi,
offload), eg.:
        pwreport --checks --include-tags scalar --verbose --config build/compile_commands.json library/aes.c

1 file successfully analyzed and 0 failures in 748 ms
```

Step 4: Automatically rewriting the source code with Codee Assistant

Codee provides source code rewriting capabilities that enable the optimization of several loops in the same function in one single invocation of the *pwdirectives* tool. The report shown below optimizes the 5 loops of the *mbedtls_aes_crypt_xts* function reported in Auto mode in the screening report (flag *--auto*).

```
$ pwdirectives --auto --vector omp --in-place --config build/compile_commands.json library/aes.c:mbedtls_aes_crypt_xts
[C] target compiler '/opt/cray/pe/gcc/11.2.0/bin/gcc', 11.2.0
  Full version name: gcc version 11.2.0 20210728 (Cray Inc.) (GCC)
  Optimization flags for rewriting: -O2
  Optimization flags for verification: -O2 -fopenmp-simd

pwdirectives: warning: remember to add the -fopenmp-simd flag to the compiler flags needed to compile this file.
Otherwise, the compiler will not be able to vectorize the loops annotated with pragmas by Codee.
Loop                              gcc-11.2 Codee 2023.0-29386 Verified by Codee
--------------------------------- -------- ------------------ -----------------
/global/homes/s/user/performance-demos/MbedTLS/v3.1.0/library/aes.c
|- mbedtls_aes_crypt_xts:1126:5    n/a      no (cost)
|- mbedtls_aes_crypt_xts:1127:5    n/a      no (cost)
|- mbedtls_aes_crypt_xts:1129:5    n/a      no (cost)
|- mbedtls_aes_crypt_xts:1130:5    n/a      no (cost)
|- mbedtls_aes_crypt_xts:1131:5    n/a      no (cost)
|- mbedtls_aes_crypt_xts:1147:5    n/a      no (othr)
|  |- mbedtls_aes_crypt_xts:1162:9 n/a      auto               not verified
|  `- mbedtls_aes_crypt_xts:1169:9 n/a      auto               not verified
|- mbedtls_aes_crypt_xts:1194:9    n/a      auto               not verified
|- mbedtls_aes_crypt_xts:1202:9    n/a      yes
`- mbedtls_aes_crypt_xts:1211:9    n/a      auto               not verified

Compiler command used by Codee to verify the auto-generated code:
  /opt/cray/pe/gcc/11.2.0/bin/gcc -I /global/homes/s/user/performance-demos/MbedTLS/v3.1.0/include -I
/global/homes/s/user/performance-demos/MbedTLS/v3.1.0/library -Wall -Wextra -Wwrite-strings -Wformat=2
-Wno-format-nonliteral -Wvla -Wshadow -Wmissing-declarations -Wmissing-prototypes -O2 -o
CMakeFiles/mbedcrypto_static.dir/aes.c.o -c /global/homes/s/user/performance-demos/MbedTLS/v3.1.0/library/aes.c
-fopt-info-vec-missed -fopt-info -c

Loop : loop name following the syntax <file>:<function>:<line>:<column>
gcc-11.2 : Target compiler vectorization status:
    auto: loop automatically vectorized by the compiler
    no: loop not vectorized by the compiler. Could happen for different reasons:
      no (cost): the compiler's cost model recommends so
      no (ctrl): complex control flow inhibits vectorization
      no (dep) : there is (or seems to be) a dependency inhibiting vectorization
      no (prec): potential precision loss if vectorized
      no (vgen): SIMD instruction generator not supported by the compiler
      no (outr): unsupported outer loop
      no (unrl): the loop was fully unrolled by the compiler
      no (call): the loop was replaced by a library call
      no (othr): any other reason
```

```
    n/a: no information was provided by the compiler for this loop
Codee 2023.0-29386 : 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 (‘verified: vect. by gcc-11.2’ when Codee successfully produced
an explicit omp|gcc|clang SIMD pragma that resulted in automatic compiler vectorization, ‘verified: not vectorized by
gcc-11.2’ when the compiler reports that it did not vectorize it (despite the new pragma), ‘not verified’ when no
vectorization report for this loop was recognized, and 'error' when a problem has occurred.

1 file successfully analyzed and 0 failures in 770 ms
```

A closer look at the *pwdirectives* output reveals that the *gcc* compiler reports the 5 target loops (lines 1162, 1169, 1194, 1202, 1211) as not vectorized because the compiler (message “*n/a*” indicates that *gcc* did not emit any user message related to vectorization). Instead, Codee reported the 4 original loops as vectorizable (message “*auto*”) and produced the optimized source code annotating it with OpenMP vectorization pragmas (“*#pragma omp simd*”). Note that at the beginning Codee was set up to interoperate with the *gcc-11.2* compiler (located at */opt/cray/pe/gcc/11.2.0/bin/gcc*). As a result, Codee identified the loops reported as not vectorizable by the compiler and attempted to discover new opportunities for vectorization.

Given the large size of the code, find below the details of the annotations produced by Codee in the format of a code patch:

```
@@ -1159,6 +1159,7 @@ int mbedtls_aes_crypt_xts( mbedtls_aes_xts_context *ctx,
             mbedtls_gf128mul_x_ble( tweak, tweak );
         }

+        #pragma omp simd
         for( i = 0; i < 16; i++ )
             tmp[i] = input[i] ^ tweak[i];

@@ -1166,6 +1167,7 @@ int mbedtls_aes_crypt_xts( mbedtls_aes_xts_context *ctx,
         if( ret != 0 )
             return( ret );

+        #pragma omp simd
         for( i = 0; i < 16; i++ )
             output[i] = tmp[i] ^ tweak[i];

@@ -1191,6 +1193,7 @@ int mbedtls_aes_crypt_xts( mbedtls_aes_xts_context *ctx,
          * byte of cyphertext we won't steal. At the same time, copy the
          * remainder of the input for this final round (since the loop bounds
          * are the same). */
+        #pragma omp simd lastprivate(i)
         for( i = 0; i < leftover; i++ )
         {
             output[i] = prev_output[i];
@@ -1208,6 +1211,7 @@ int mbedtls_aes_crypt_xts( mbedtls_aes_xts_context *ctx,
         /* Write the result back to the previous block, overriding the previous
          * output we copied. */
+        #pragma omp simd
         for( i = 0; i < 16; i++ )
             prev_output[i] = tmp[i] ^ t[i];
     }
```

Step 5: Compiling, running and benchmarking the optimized code

The MbedTLS software does not use OpenMP by default. The optimizations that take advantage of vectorization were implemented with OpenMP SIMD pragmas. Thus, it is necessary to include the compiler flag *-fopenmp-simd* flag [3] during the cmake build instruction of MBedTLS:

```
$ cmake -DENABLE_TESTING=On -DCMAKE_C_COMPILER=gcc -DUSE_SHARED_MBEDTLS_LIBRARY=On -DCMAKE_BUILD_TYPE=Release
-DCMAKE_EXPORT_COMPILE_COMMANDS=1 -DCMAKE_C_FLAGS=-fopenmp-simd -DMBEDTLS_FATAL_WARNINGS=Off -B buildVec -G "Unix
Makefiles

$ make -C buildVec/
```

Now the optimized source file "*aes.c*" is ready to be compiled, run and benchmarked:

```
$ buildVec/programs/test/benchmark aes_xts

  AES-XTS-128             :     665714 KiB/s,          3 cycles/byte
  AES-XTS-256             :     556754 KiB/s,          4 cycles/byte
```

In order to measure the performance gain achieved by using Codee, let's run the previous compiled version at *build/*:

```
$ build/programs/test/benchmark aes_xts

  AES-XTS-128             :     502229 KiB/s,          4 cycles/byte
  AES-XTS-256             :     449053 KiB/s,          5 cycles/byte
```

Finally, note that a performance gain of +20% faster has been achieved by following the performance optimization best practices recommended by Codee. This indicates that Codee is a good complement to the compiler and enables the programmer to create performant code.

[3] Note: The *-fopenmp* flag also works in this case. However, we recommend *-fopenmp-simd* if we are only using SIMD pragmas because it does not include references to the OpenMP libraries for multithreading and offloading, and in this case we are taking advantage of vectorization only.

Step 6. Further optimize MBedTLS exploring other functions

The process described above for one function of MBedTLS can be repeated for other functions identified in the Codee Screening Report. For illustrative purposes, consider the following *pwdirectives* invocations for the source code files "*aes.c*", "*cmac.c*" and "*aria.c*".

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

```
$ pwdirectives --auto --vector omp --in-place --config build/compile_commands.json library/cmac.c:cmac_xor_block
```

```
$ pwdirectives --auto --vector omp --in-place --config build/compile_commands.json
library/aria.c:mbedtls_aria_crypt_cbc
```

Finally, note that Codee Coding Assistant capabilities also enable automation of performance optimizations in CI/CD pipelines and contribute to improving the maintainability of codes by automating source code rewriting from the same original source code. For example, starting from a single source file, Codee can automatically generate different versions using vectorization pragmas for different compilers. For illustrative purposes, consider the following *pwdirectives* commands that annotate the source code either with OpenMP SIMD pragmas, *gcc* SIMD pragmas or *clang* SIMD pragmas. The *-o* flag allows us to generate the optimized version of the code in a new output file.

```
$ pwdirectives --auto --vector omp --config build/compile_commands.json  -o library/aes_omp.c
library/aes.c:mbedtls_aes_crypt_cbc
```

```
$ pwdirectives --auto --vector gcc --config build/compile_commands.json  -o library/aes_gcc.c
library/aes.c:mbedtls_aes_crypt_cbc
```

```
$ pwdirectives --auto --vector clang --config build/compile_commands.json -o library/aes_clang.c
library/aes.c:mbedtls_aes_crypt_cbc
```

Overall, Codee is a good complement to the compiler and enables the programmer to create performant code. Codee supports an iterative development process, so that the performance optimizations identified and tested can be automatically applied to the source code. This has an impact on the maintainability and productivity of the developers.

Step 7. Submitting a job to a CPU node

Pelmutter has CPU and GPU nodes. The user is expected to follow this step-by-step guide, typically running the commands in an interactive session. Additionally, a set of scripts to do the benchmarking using the queueing system of Perlmutter are provided.

For illustrative purposes, the user is expected to run the following *sbatch* command:

```
$ sbatch launch.sh
```

The script "*launch.sh*" to run a job in a CPU node is as follows [4]:

```
#!/bin/bash
#SBATCH -A ntrain8
#SBATCH --reservation=codee_day1_cpu
#SBATCH -C cpu
#SBATCH -J Codee_MbedTLS
#SBATCH -q regular
#SBATCH -t 0:20:00
#SBATCH -N 1
#SBATCH --ntasks=1          #Indicate the maximum number of processes
#SBATCH --ntasks-per-node=1 #Indicate how many tasks per node you want to run
#SBATCH --cpus-per-task=32  #Indicate how many CPU cores per task you need

export SLURM_CPU_BIND="cores"
export OMP_NUM_THREADS=32

srun -N 1 -n 1 MbedTLS.sh
```

Finally, the script *MbedTLS.sh* to build and run all the codes is as follows (note it does not include the generation of new code versions with Codee *pwdirectives*):

[4] Note that the *--reservation* named "*codee_day1*" needs to be renamed as "*codee_day2*" or as any other reservation available to the user.

```
#!/bin/bash

function printRunCommand(){
    ## Print the command
    printf "\n$ $@\n"
    ## Run the command
    $@
}

module load PrgEnv-gnu
module load codee

rm -rf build/ buildVec/
cp library/aes_original.c library/aes.c

echo ""
echo ""
echo "Build Serial"
printRunCommand "cmake -DENABLE_TESTING=On -DCMAKE_C_COMPILER=gcc -DUSE_SHARED_MBEDTLS_LIBRARY=On
-DCMAKE_BUILD_TYPE=Release -DCMAKE_EXPORT_COMPILE_COMMANDS=1 -DMBEDTLS_FATAL_WARNINGS=Off -B build"
printRunCommand "make -C build"

echo ""
echo ""
echo "Vectorize"
printRunCommand "pwdirectives --auto --vector omp --in-place --config build/compile_commands.json
library/aes.c:mbedtls_aes_crypt_xts"

echo ""
echo ""
echo "Build Vectorized"
printRunCommand "cmake -DENABLE_TESTING=On -DCMAKE_C_COMPILER=gcc -DUSE_SHARED_MBEDTLS_LIBRARY=On
-DCMAKE_BUILD_TYPE=Release -DCMAKE_EXPORT_COMPILE_COMMANDS=1 -DCMAKE_C_FLAGS=-fopenmp-simd
-DMBEDTLS_FATAL_WARNINGS=Off -B buildVec"
printRunCommand "make -C buildVec"

echo ""
echo ""
echo "Run Serial"
printRunCommand "build/programs/test/benchmark aes_xts"

echo ""
echo ""
echo "Build Vectorized"
printRunCommand "buildVec/programs/test/benchmark aes_xts"
```