codee

Quickstart Guide of Codee Matrix-Matrix Multiplication (MATMUL-C) Step-by-Step in Perlmutter @NERSC

Step 1. Setup the environment

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

```
$ module load codee
$ module load PrgEnv-nvidia
```

Run Codee configuration wizard to set up the compiler used to generate executables. In this case, select the option "*No target compiler (Skip)*" [2], meaning that Codee will not attempt to interpret the user messages emitted by the compiler.

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

Step 2: Understanding the implementation of MATMUL

The implementation of the Matrix-Matrix multiplication code in the C programming language considered in this case is shown below [3]. Note it is a standard implementation using dynamically allocated arrays of floating-point values. The two-dimensional arrays are traversed in the naive row-major order, iterating per row and later iterating per column within the row (www.codee.com/knowledge/glossary-row-major-and-column-major-order/).

[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.
[3] Note the sources of the source codes used in the NERSC course are distributed as a ZIP file, which creates the folder *nersc_examples_2023/* once uncompressed.

```
 5   // C (m x n) = A (m x p) * B (p x n)
 6   void matmul(size_t m, size_t n, size_t p, double **A, double **B, double **C) {
 7       // Initialization
 8       for (size_t i = 0; i < m; i++) {
 9           for (size_t j = 0; j < n; j++) {
10               C[i][j] = 0;
11           }
12       }
13
14       // Accumulation
15       for (size_t i = 0; i < m; i++) {
16           for (size_t j = 0; j < n; j++) {
17               for (size_t k = 0; k < p; k++) {
18                   C[i][j] += A[i][k] * B[k][j];
19               }
20           }
21       }
22   }
```

Step 3: Producing the Codee Screening Report (optional)

The Codee Screening Report is intended to help assess the speed-up potential of the code. Invoke the *pwreport --screening* command enabling source code checks for GPU (*--screening-tags all*), passing the target source code (*main.c*) and its corresponding compilation flags after the special mark "*--*" (*-I include -fast*). In this case, the screening reported 8 checks in the source code, the breakdown being 1 check related to scalar optimizations, 3 checks related to memory issues, 2 checks related to vectorization, 1 check related to multithreading and 1 check related to offloading.

```
$ pwreport --screening main.c --include-tags all -- -I include -fast
Compiler flags: -I include -fast

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

SCREENING REPORT

Target Lines of code Optimizable lines Analysis time # checks Effort Cost     Profiling
------ ------------- ----------------- ------------- -------- ------ ------- ---------
main.c 55            14                137 ms        8        57 h   1865€   n/a
------ ------------- ----------------- ------------- -------- ------ ------- ---------
Total  55            14                137 ms        8        57 h   1865€   n/a

CHECKS PER STAGE OF THE PERFORMANCE OPTIMIZATION ROADMAP
Target Scalar Control Memory Vector Multi Offload
------ ------ ------- ------ ------ ----- -------
main.c 1      0       3      2           1       1
------ ------ ------- ------ ------ ----- -------
Total  1      0       3      2           1       1

TOTAL NUMBER OF LOOPS FOR CODEE AUTO AND GUIDED MODES
                  ---------------- # checks ----------------
Codee mode # Loops Scalar Control Memory Vector Multi Offload
---------- ------- ------ ------- ------ ------ ----- -------
Auto       1       0      0       0      1      0     0
Guided     3       0      0       3      1      1     1

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 main.c --include-tags all -- -I include -fast

  You can specify multiple inputs which will be displayed as multiple rows (ie. targets) in the table, eg:
         pwreport --screening some/other/dir main.c --include-tags all -- -I include -fast

  Use --checks to find out details about the detected checks:
         pwreport --checks main.c --include-tags all -- -I include -fast

  You can automatically vectorize every vectorizable loop of one function with:
         pwdirectives --auto --simd omp --in-place main.c -- -I include -fast

  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 main.c -- -I include -fast

1 file successfully analyzed and 0 failures in 138 ms
```

Step 4: Producing the Codee Checks Report

The Codee output suggests subsequent Codee commands useful to dig deeper into the performance issues discovered by the tool. The suggested next step is to produce the Codee Checks Report to list all the checks found in the code, by invoking the *pwreport --checks* command. The output format is similar to other static code analyzers in order to facilitate the user uptake and integration in the development workflow. The details about the checks reported in the screening phase are shown, including the position where the checks were detected in the source code and the title of each check.

```
$ pwreport --checks main.c:matmul --include-tags all -- -I include -fast
Compiler flags: -I include -fast

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

CHECKS REPORT

main.c:18:17 [PWR048]: Replace multiplication/addition combo with an explicit call to fused multiply-add
main.c:9:9 [PWR053]: consider applying vectorization to forall loop
main.c:15:5 [PWR035]: avoid non-consecutive array access for variables 'A', 'B' and 'C' to improve performance
main.c:15:5 [PWR050]: consider applying multithreading parallelism to forall loop
main.c:15:5 [PWR055]: consider applying offloading parallelism to forall loop
main.c:16:9 [PWR039]: consider loop interchange to improve the locality of reference and enable vectorization
main.c:17:13 [PWR010]: 'B' multi-dimensional array not accessed in row-major order
main.c:17:13 [RMK010]: 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:
         pwreport --verbose --checks main.c:matmul --include-tags all -- -I include -fast

  More details on the defects, recommendations and more in the Knowledge Base:
         https://www.codee.com/knowledge/

1 file successfully analyzed and 0 failures in 135 ms
```

Codee suggests to request further information by invoking the *pwreport --checks* command with verbosity enabled (*--verbose*). In the scope of this document, we remark a memory inefficiency that can be fixed through loop interchange (PWR039) and a loop that is a candidate to be offloaded to the GPU (PWR055).

```
$ pwreport --verbose --checks main.c:matmul --include-tags all -- -I include -fast
Compiler flags: -I include -fast

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

CHECKS REPORT

main.c:18:17 [PWR048]: Replace multiplication/addition combo with an explicit call to fused multiply-add
   Suggestion: Replace a combination of multiplication and addition `C[i][j] += A[i][k] * B[k][j];`, with a call to the
`fma` function.
  Documentation: https://www.codee.com/knowledge/pwr048

main.c:9:9 [PWR053]: consider applying vectorization to forall loop
  Suggestion: use pwdirectives to automatically optimize the code
  Documentation: https://www.codee.com/knowledge/pwr053
  AutoFix:
    * Using OpenMP pragmas (recommended):
        pwdirectives --vector omp --in-place main.c:9:9 -- -I include -fast
    * Using Clang compiler pragmas:
        pwdirectives --vector clang --in-place main.c:9:9 -- -I include -fast
    * Using GCC pragmas
        pwdirectives --vector gcc --in-place main.c:9:9 -- -I include -fast
    * Using ICC pragmas:
        pwdirectives --vector icc --in-place main.c:9:9 -- -I include -fast

main.c:15:5 [PWR035]: avoid non-consecutive array access for variables 'A', 'B' and 'C' to improve performance
  Non-consecutive array access:
    18:                 C[i][j] += A[i][k] * B[k][j];
   Suggestion: consider using techniques like loop fusion, loop interchange, loop tiling or changing the data layout to
avoid non-sequential access to variables 'A', 'B' and 'C'.
  Documentation: https://www.codee.com/knowledge/pwr035

main.c:15:5 [PWR050]: consider applying multithreading parallelism to forall loop
  Suggestion: use pwdirectives to automatically optimize the code
  Documentation: https://www.codee.com/knowledge/pwr050
  AutoFix (choose one option):
      * Using OpenMP 'for' (recommended):
        pwdirectives --multi omp-for --in-place main.c:15:5 -- -I include -fast
      * Using OpenMP 'taskwait':
        pwdirectives --multi omp-taskwait --in-place main.c:15:5 -- -I include -fast
      * Using OpenMP 'taskloop':
        pwdirectives --multi omp-taskloop --in-place main.c:15:5 -- -I include -fast

main.c:15:5 [PWR055]: consider applying offloading parallelism to forall loop
  Suggestion: use pwdirectives to automatically optimize the code
  Documentation: https://www.codee.com/knowledge/pwr055
  AutoFix:
    * Using OpenMP (recommended):
        pwdirectives --offload omp-teams --in-place main.c:15:5 -- -I include -fast
    * Using OpenAcc:
        pwdirectives --offload acc --in-place main.c:15:5 -- -I include -fast

main.c:16:9 [PWR039]: 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: loop interchange can be used to improve the performance of the loop nest.
  Documentation: https://www.codee.com/knowledge/pwr039
  AutoFix:
    pwdirectives --memory loop-interchange --in-place main.c:16:9 -- -I include -fast

main.c:17:13 [PWR010]: 'B' multi-dimensional array not accessed in row-major order
  Accesses:
    18:                 C[i][j] += A[i][k] * B[k][j];
  Suggestion: change the code to access the 'B' multi-dimensional array in a row-major order
  Documentation: https://www.codee.com/knowledge/pwr010

main.c:17:13 [RMK010]: the vectorization cost model states the loop is not a SIMD opportunity due to strided memory
accesses in the loop body
```

```
  Documentation: https://www.codee.com/knowledge/rmk010

SUGGESTIONS

  More details on the defects, recommendations and more in the Knowledge Base:
         https://www.codee.com/knowledge/

1 file successfully analyzed and 0 failures in 110 ms
```

Step 5: Generating code optimized for single-core (loop interchange)

The Codee software also includes source code rewriting capabilities to implement a performance optimization triggered by the Codee Static Code Analyzer. The source code rewriting is always triggered by the developer through the invocation of the *pwdirectives* tool, it never happens automatically without the supervision of the developer. Several examples of usage of the Codee Coding Assistant are shown below.

First, focus on the PWR039 check reported by Codee related to a memory inefficiency that can be fixed by implementing loop interchange in the source code. As shown above, the verbose Codee Checks Report for PWR039 is as follows:

```
...
main.c:16:9 [PWR039]: 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: loop interchange can be used to improve the performance of the loop nest.
  Documentation: https://www.codee.com/knowledge/pwr039
  AutoFix:
    pwdirectives --memory loop-interchange --in-place main.c:16:9 -- -I include -fast
...
```

Second, produce a new optimized source code (new file name "*matmul_li.c*") that implements the loop interchange recommendation (PWR039). For this purpose copy and paste the *pwdirectives* invocation suggested by the tool in the AutoFix section [4].

```
$ pwdirectives --memory loop-interchange main.c:16:9 -o main_li.c -- -I include -fast
Compiler flags: -I include -fast

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

Results for file 'main.c':
  Successfully loop interchange at main.c:16:9:
Successfully created main_li.c
```

[4] Note that in order to reproduce this testing in a GPU node, a new output file is created by replacing the "*--in-place*" flag of the Autofix suggestion by "*-o main_li.c*".

Third, review the optimized source code generated by the *pwdirectives* tool, which actually interchanged the headers of the loops located at the lines 16 and 17. The new code looks like this:

```
 5   // C (m x n) = A (m x p) * B (p x n)
 6   void matmul(size_t m, size_t n, size_t p, double **A, double **B, double **C) {
 7       // Initialization
 8       for (size_t i = 0; i < m; i++) {
 9           for (size_t j = 0; j < n; j++) {
10               C[i][j] = 0;
11           }
12       }
13
14       // Accumulation
15       for (size_t i = 0; i < m; i++) {
16           for (size_t k = 0; k < p; k++) {
17               for (size_t j = 0; j < n; j++) {
18                   C[i][j] += A[i][k] * B[k][j];
19               }
20           }
21       }
22   }
```

Finally, let's benchmark the original source code ("*main.c*") and the optimized version based on loop interchange ("*main_li.c*"). As shown below, the single-core performance gain is 3.2x using the Nvidia *nvc* compiler with maximum optimization level (*-fast*), reducing the runtime from 39.6 seconds down to 12.2 seconds.

```
$ nvc -fast matrix.c clock.c main.c -o matmul -I include
$ ./matmul 3000
- Input parameters
n       = 3000
- Executing test...
time (s)= 39.637778
size    = 3000
chksum  = 546933812237

$ nvc -fast matrix.c clock.c main_li.c -o matmul_li -I include
$ ./matmul_li 3000
- Input parameters
n       = 3000
- Executing test...
time (s)= 12.204002
size    = 3000
chksum  = 546933812237
```

Step 6: Generating code optimized for GPU (loop interchange + offloading)

Let's use Codee to generate GPU-enabled code using OpenACC and OpenMP compiler pragmas. The starting point is the Codee Checks Report corresponding to the MATMUL version with loop interchange (file name "*main_li.c*"), which points out a PWR055:

```
...
main.c:15:5 [PWR055]: consider applying offloading parallelism to forall loop
  Suggestion: use pwdirectives to automatically optimize the code
  Documentation: https://www.codee.com/knowledge/pwr055
  AutoFix:
    * Using OpenMP (recommended):
        pwdirectives --offload omp-teams --in-place main.c:15:5 -- -I include -fast
    * Using OpenAcc:
        pwdirectives --offload acc --in-place main.c:15:5 -- -I include -fast
...
```

First, produce a further optimized source code (new file name "*main_li_acc.c*") that implements the offload opportunity (PWR055) using the OpenACC directives on top of the code version with loop interchange. For this purpose copy and paste the *pwdirectives* invocation suggested by the tool in the AutoFix section [5]:

```
$ pwdirectives --offload acc -o main_li_acc.c main_li.c:15:5 -- -I include -fast
Compiler flags: -I include -fast

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

Results for file 'main_li.c':
  Successfully parallelized loop at 'main_li.c:matmul:15:5' [using offloading without teams]:
      [INFO] main_li.c:15:5 Parallel forall: variable 'C'
      [INFO] main_li.c:15:5 Parallel region defined by OpenACC directive 'parallel'
      [INFO] main_li.c:15:5 Loop parallelized with OpenACC directive 'loop'
      [INFO] main_li.c:15:5 Data region for host-device data transfers defined by OpenACC directive 'data'
      [INFO] main_li.c:15:5 Make sure there is no aliasing among variables: A, B, C
Successfully created main_li_acc.c

Minimum software stack requirements: OpenACC version 2.0 with offloading capabilities
```

Thus, the resulting MATMUL version that implements loop interchange and OpenACC offload is as follows:

```
 5   // C (m x n) = A (m x p) * B (p x n)
 6   void matmul(size_t m, size_t n, size_t p, double **A, double **B, double **C) {
 7       // Initialization
 8       for (size_t i = 0; i < m; i++) {
 9           for (size_t j = 0; j < n; j++) {
10               C[i][j] = 0;
11           }
12       }
13
14       // Accumulation
15       #pragma acc data copyin(A[0:m][0:p], B[0:p][0:n], m, n, p) copy(C[0:m][0:n])
16       #pragma acc parallel
17       #pragma acc loop
```

[5] Note that in order to reproduce this testing in a GPU node, a new output file is created by replacing the "*--in-place*" flag of the Autofix suggestion by "*-o main_li_acc.c*".

```
    18        for (size_t i = 0; i < m; i++) {
    19            for (size_t k = 0; k < p; k++) {
    20                for (size_t j = 0; j < n; j++) {
    21                    C[i][j] += A[i][k] * B[k][j];
    22                }
    23            }
    24        }
    25    }
```

Second, use Codee to generate the corresponding version based on OpenMP offload pragmas [6]:

```
$ pwdirectives --offload omp-teams -o main_li_omp.c main_li.c:15:5 -- -I include -fast
Compiler flags: -I include -fast

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

Results for file 'main_li.c':
  Successfully parallelized loop at 'main_li.c:matmul:15:5' [using offloading]:
      [INFO] main_li.c:15:5 Parallel forall: variable 'C'
      [INFO] main_li.c:15:5 Loop parallelized with teams using OpenMP directive 'target teams distribute parallel for'
Successfully created main_li_omp.c

Minimum software stack requirements: OpenMP version 4.5 with offloading capabilities
```

Thus, the resulting MATMUL version that implements loop interchange and OpenMP offload is as follows:

```
     5    // C (m x n) = A (m x p) * B (p x n)
     6    void matmul(size_t m, size_t n, size_t p, double **A, double **B, double **C) {
     7        // Initialization
     8        for (size_t i = 0; i < m; i++) {
     9            for (size_t j = 0; j < n; j++) {
    10                C[i][j] = 0;
    11            }
    12        }
    13
    14        // Accumulation
    15        #pragma omp target enter data map(to: A[0:m])
    16        for(int i0 = 0; i0 < m; ++i0) {
    17          #pragma omp target enter data map(to: A[i0][0:p])
    18        }
    19        #pragma omp target enter data map(to: B[0:p])
    20        for(int i0 = 0; i0 < p; ++i0) {
    21          #pragma omp target enter data map(to: B[i0][0:n])
    22        }
    23        #pragma omp target enter data map(to: C[0:m])
    24        for(int i0 = 0; i0 < m; ++i0) {
    25          #pragma omp target enter data map(to: C[i0][0:n])
    26        }
    27        #pragma omp target teams distribute parallel for shared(A, B, m, n, p) map(to: m, n, p) schedule(static)
    28        for (size_t i = 0; i < m; i++) {
    29            for (size_t k = 0; k < p; k++) {
    30                for (size_t j = 0; j < n; j++) {
    31                    C[i][j] += A[i][k] * B[k][j];
    32                }
    33            }
    34        }
    35        for(int i0 = 0; i0 < m; ++i0) {
    36          #pragma omp target exit data map(from: C[i0][0:n])
    37        }
    38        #pragma omp target exit data map(from: C[0:m])
    39    }
```

Finally, let's benchmark the original source code ("*main.c*") versus the optimized version based on loop interchange and GPU offload directives, both OpenACC offload ("*main_li_acc.c*") and OpenMP

[6] Note that in order to reproduce this testing in a GPU node, a new output file is created by replacing the "*--in-place*" flag of the Autofix suggestion by "*-o main_li_omp.c*".

offload ("*main_li_omp.c*"). As shown below, the GPU-enabled performance gain is 6.4x and 7.8x using the Nvidia *nvc* compiler with maximum optimization level (*-fast*), reducing the runtime from 39.6 seconds down to 6.1 and 5.1 seconds, respectively.

```
$ nvc -fast matrix.c clock.c main.c -o matmul -I include
$ ./matmul 3000
- Input parameters
n       = 3000
- Executing test...
time (s)= 39.637778
size    = 3000
chksum  = 546933812237

$ nvc -fast -acc -target=gpu -Minfo=acc matrix.c clock.c main_li_acc.c -o matmul_li_acc -I include
$ ./matmul_li_acc 3000
- Input parameters
n       = 3000
- Executing test...
time (s)= 6.198646
size    = 3000
chksum  = 546933812237

$ nvc -fast -mp -target=gpu -Minfo=mp matrix.c clock.c main_li_omp.c -o matmul_li_omp -I include
$ ./matmul_li_omp 3000
- Input parameters
n       = 3000
- Executing test...
time (s)= 5.081679
size    = 3000
chksum  = 546933812237
```

Step 7. Submitting a job to a GPU 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 GPU node is as follows [7]:

```
#!/bin/bash
#SBATCH -A ntrain8
#SBATCH --reservation=codee_day1_gpu
#SBATCH -C gpu
#SBATCH -J Codee_Matmul_C
#SBATCH -q regular
#SBATCH -t 0:10:00
#SBATCH -N 1
#SBATCH --ntasks-per-node=1
#SBATCH -c 128
#SBATCH --gpus-per-task=1

export SLURM_CPU_BIND="cores"
srun MATMUL_C.sh
```

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

Finally, the script *MATMUL_C.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*):

```
#!/bin/bash

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

module load PrgEnv-nvidia

rm -f matmul matmul_li matmul_li_acc matmul_li_omp *.o

echo ""
echo ""
echo "Matmul"
printRunCommand "nvc -fast matrix.c clock.c main.c -o matmul -I include"
printRunCommand "./matmul 3000"

echo ""
echo ""
echo "Matmul with loop interchange"
printRunCommand "nvc -fast matrix.c clock.c main_li.c -o matmul_li -I include"
printRunCommand "./matmul_li 3000"

echo ""
echo ""
echo "Matmul with loop interchange and OpenACC"
printRunCommand "nvc -fast -acc -target=gpu -Minfo=acc matrix.c clock.c main_li_acc.c -o matmul_li_acc -I include"
printRunCommand "./matmul_li_acc 3000"

echo ""
echo ""
echo "Matmul with loop interchange and OpenMP"
printRunCommand "nvc -fast -mp -target=gpu -Minfo=mp matrix.c clock.c main_li_omp.c -o matmul_li_omp -I include"
printRunCommand "./matmul_li_omp 3000"
```

Additional remarks about the optimization of MATMUL

Codee helps enforce performance optimization best practices. Following Codee hints, the source code created by the developer achieves a performance that is less dependent on the capabilities of the target compiler [8]. It is good practice that the developer writes code where the performance optimizations are explicit in the source code. This makes the source code of the application more compiler-friendly and hardware-friendly from the point of view of performance.

About the Nvidia compiler in Perlmutter

Regarding the MATMUL sequential source code, it is noticeable that the Nvidia *nvc* compiler: (1) does not apply loop interchange to the C source code, and (2) reports attempts to vectorize loops but does not report opportunities in loops that might be offloaded to the GPU. Same remarks apply to the OpenACC-enabled source code written in C. The user messages reported by the *nvc* compiler are shown below, indicating that Codee is a good complement to the compiler and enables the programmer to create performant code for the GPU.

```
$ nvc -fast  matrix.c clock.c main.c -o matmul -I include -Minfo=all
matrix.c:
new_matrix:
     23, Loop not vectorized: may not be beneficial for target
         Loop not vectorized: unknown
         Loop unrolled 4 times
rand_matrix:
     43, Loop not vectorized/parallelized: contains call
checksum_matrix:
     55, Generated vector simd code for the loop containing reductions
clock.c:
getClock:
      9, FMA (fused multiply-add) instruction(s) generated
main.c:
matmul:
      8, Loop not fused: dependence chain to sibling loop
      9, Generated vector simd code for the loop
     17, Loop not vectorized: data dependency
         Loop unrolled 2 times
         FMA (fused multiply-add) instruction(s) generated
     18, FMA (fused multiply-add) instruction(s) generated
main:
     58, Loop not fused: function call before adjacent loop
     59, matmul inlined, size=17 (inline) file main.c (6)
           8, Loop not fused: different controlling conditions
           9, Generated vector simd code for the loop
          17, Loop not vectorized: data dependency
              Loop unrolled 2 times
              FMA (fused multiply-add) instruction(s) generated
          58, Loop not vectorized/parallelized: too deeply nested
```

[8] Documentation about the compilers available in Perlmutter https://docs.nersc.gov/development/compilers/base/ .

```
$ nvc -acc -target=gpu -Minfo=all -fast matrix.c clock.c main_li_acc.c -o matmul_li_acc  -I include
matrix.c:
new_matrix:
     23, Loop not vectorized: may not be beneficial for target
         Loop not vectorized: unknown
         Loop unrolled 4 times
rand_matrix:
     43, Loop not vectorized/parallelized: contains call
checksum_matrix:
     55, Generated vector simd code for the loop containing reductions
clock.c:
getClock:
      9, FMA (fused multiply-add) instruction(s) generated
main_li_acc.c:
matmul:
      8, Loop not fused: function call before adjacent loop
      9, Generated vector simd code for the loop
     16, Generating copyin(A[:m][:n]) [if not already present]
         Generating copy(C[:m][:p]) [if not already present]
         Generating copyin(m,n,p,B[:n][:p]) [if not already present]
     18, Generating NVIDIA GPU code
         20, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         21, #pragma acc loop seq
         22, #pragma acc loop seq
     21, Complex loop carried dependence of A->-> prevents parallelization
         Loop carried dependence of C->-> prevents parallelization
         Loop carried backward dependence of C->-> prevents vectorization
         Complex loop carried dependence of C->->,B->-> prevents parallelization
     22, Complex loop carried dependence of A->->,C->->,B->-> prevents parallelization
main:
     65, Loop not vectorized/parallelized: contains call
```

```
$ nvc -mp -target=gpu -Minfo=all -fast  matrix.c clock.c main_li_omp.c -o matmul_li_omp  -I include
matrix.c:
new_matrix:
     23, Loop not vectorized: may not be beneficial for target
         Loop not vectorized: unknown
         Loop unrolled 4 times
rand_matrix:
     43, Loop not vectorized/parallelized: contains call
checksum_matrix:
     55, Generated vector simd code for the loop containing reductions
clock.c:
main_li_omp.c:
matmul:
      8, Loop not fused: function call before adjacent loop
      9, Generated vector simd code for the loop
     12, Generating target enter data map(to: A[:m])
     16, Loop not vectorized/parallelized: contains call
     18, Generating target enter data map(to: B[:p],A[i0][:p])
     20, Loop not vectorized/parallelized: contains call
     22, Generating target enter data map(to: C[:m],B[i0][:n])
     24, Loop not vectorized/parallelized: contains call
     26, #omp target teams distribute parallel for
         26, Generating "nvkernel_matmul_F1L26_2" GPU kernel
         28, Loop parallelized across teams and threads(128), schedule(static)
     26, Generating target enter data map(to: C[i0][:n])
         Generating map(to:m,p,n)
     28, Loop not vectorized/parallelized: not countable
     30, Loop not vectorized: data dependency
         Loop unrolled 2 times
         FMA (fused multiply-add) instruction(s) generated
     31, FMA (fused multiply-add) instruction(s) generated
     35, Loop not vectorized/parallelized: contains call
     37, Generating target exit data map(from: C[i0][:n])
     39, Generating target exit data map(from: C[:m])
main:
     75, Loop not vectorized/parallelized: contains call
/usr/bin/ld: warning: /tmp/pgcudafatG6ofGldlIIjz.o: missing .note.GNU-stack section implies executable stack
/usr/bin/ld: NOTE: This behaviour is deprecated and will be removed in a future version of the linker
```

About the GNU compiler in Perlmutter

Regarding the GNU gcc compiler, the same remarks apply to the MATMUL sequential source code written in C. Regarding the OpenACC-enabled, the gcc compiler does not provide OpenACC support mature enough to generate efficient code for the GPU. For illustrative purposes, see the messages reported by *gcc* set up in maximum optimization level (*-fast*) and in verbose mode (*-fopt-info-all*):

```
$ module use /global/cfs/cdirs/m1759/yunhe/Modules/perlmutter/modulefiles
$ module load gcc/12.1.0

$ gcc -I include main.c -fopt-info-all -c -Ofast
main.c:59:9: note: Considering inline candidate matmul/22.
main.c:59:9: missed:   will not early inline: main/23->matmul/22, call is cold and code would grow at least by 32
main.c:76:5: missed:   not inlinable: main/23 -> delete_matrix/31, function body not available
main.c:75:5: missed:   not inlinable: main/23 -> delete_matrix/31, function body not available
main.c:74:5: missed:   not inlinable: main/23 -> delete_matrix/31, function body not available
main.c:69:5: missed:   not inlinable: main/23 -> printf/24, function body not available
main.c:68:5: missed:   not inlinable: main/23 -> printf/24, function body not available
main.c:67:5: missed:   not inlinable: main/23 -> printf/24, function body not available
main.c:66:23: missed:   not inlinable: main/23 -> checksum_matrix/30, function body not available
main.c:63:26: missed:   not inlinable: main/23 -> getClock/29, function body not available
main.c:59:9: missed:   not inlinable: main/23 -> matmul/22, --param max-inline-insns-auto limit reached
main.c:55:25: missed:   not inlinable: main/23 -> getClock/29, function body not available
main.c:54:5: missed:   not inlinable: main/23 -> __builtin_puts/25, function body not available
main.c:51:5: missed:   not inlinable: main/23 -> rand_matrix/28, function body not available
main.c:50:5: missed:   not inlinable: main/23 -> rand_matrix/28, function body not available
main.c:45:9: missed:   not inlinable: main/23 -> printf/24, function body not available
main.c:43:24: missed:   not inlinable: main/23 -> new_matrix/27, function body not available
main.c:42:24: missed:   not inlinable: main/23 -> new_matrix/27, function body not available
main.c:41:24: missed:   not inlinable: main/23 -> new_matrix/27, function body not available
main.c:37:5: missed:   not inlinable: main/23 -> printf/24, function body not available
main.c:36:5: missed:   not inlinable: main/23 -> __builtin_puts/25, function body not available
main.c:35:5: missed:   not inlinable: main/23 -> sscanf/26, function body not available
main.c:29:9: missed:   not inlinable: main/23 -> __builtin_puts/25, function body not available
main.c:28:9: missed:   not inlinable: main/23 -> printf/24, function body not available
Unit growth for small function inlining: 134->134 (0%)

Inlined 0 calls, eliminated 0 functions

main.c:18:25: missed: failed: evolution of base is not affine.
main.c:18:42: missed: failed: evolution of base is not affine.
main.c:18:42: missed: failed: evolution of base is not affine.
consider run-time aliasing test between *_2 and *_5
main.c:9:30: optimized: Loop 5 distributed: split to 0 loops and 1 library calls.
main.c:18:25: missed: failed: evolution of base is not affine.
main.c:18:42: missed: failed: evolution of base is not affine.
main.c:15:26: missed: couldn't vectorize loop
main.c:15:26: missed: not vectorized: multiple nested loops.
main.c:16:30: missed: couldn't vectorize loop
main.c:18:25: missed: not vectorized: no vectype for stmt: pretmp_80 = *_10;
 scalar_type: double
main.c:17:34: missed: couldn't vectorize loop
main.c:18:32: missed: not vectorized: no vectype for stmt: _16 = *_15;
 scalar_type: double
main.c:8:26: missed: couldn't vectorize loop
main.c:10:21: missed: statement clobbers memory: __builtin_memset (_3, 0, _70);
main.c:6:6: note: vectorized 0 loops in function.
main.c:10:21: missed: statement clobbers memory: __builtin_memset (_3, 0, _70);
main.c:18:42: missed: failed: evolution of base is not affine.
main.c:10:21: missed: statement clobbers memory: __builtin_memset (_3, 0, _70);
main.c:22:1: note: ***** Analysis failed with vector mode V2DI
main.c:22:1: note: ***** Skipping vector mode V16QI, which would repeat the analysis for V2DI
main.c:24:5: note: vectorized 0 loops in function.
main.c:35:5: missed: statement clobbers memory: sscanf (_2, "%d", &param_n);
main.c:36:5: missed: statement clobbers memory: __builtin_puts (&"- Input parameters"[0]);
main.c:37:5: missed: statement clobbers memory: printf ("n\t= %i\n", param_n.0_3);
main.c:41:24: missed: statement clobbers memory: in1_mat_24 = new_matrix (rows_22, rows_22);
main.c:42:24: missed: statement clobbers memory: in2_mat_26 = new_matrix (rows_22, rows_22);
main.c:43:24: missed: statement clobbers memory: out_mat_28 = new_matrix (rows_22, rows_22);
main.c:50:5: missed: statement clobbers memory: rand_matrix (in1_mat_24, rows_22, rows_22);
main.c:51:5: missed: statement clobbers memory: rand_matrix (in2_mat_26, rows_22, rows_22);
```

```
main.c:54:5: missed: statement clobbers memory: __builtin_puts (&"- Executing test..."[0]);
main.c:55:25: missed: statement clobbers memory: time_start_33 = getClock ();
main.c:59:9: missed: statement clobbers memory: matmul (rows_22, rows_22, rows_22, in1_mat_24, in2_mat_26,
out_mat_28);
main.c:63:26: missed: statement clobbers memory: time_finish_35 = getClock ();
main.c:66:23: missed: statement clobbers memory: checksum_37 = checksum_matrix (out_mat_28, rows_22, rows_22);
main.c:67:5: missed: statement clobbers memory: printf ("time (s)= %.6f\n", _9);
main.c:68:5: missed: statement clobbers memory: printf ("size\t= %i\n", param_n.4_10);
main.c:69:5: missed: statement clobbers memory: printf ("chksum\t= %.0f\n", checksum_37);
main.c:74:5: missed: statement clobbers memory: delete_matrix (in1_mat_24);
main.c:75:5: missed: statement clobbers memory: delete_matrix (in2_mat_26);
main.c:76:5: missed: statement clobbers memory: delete_matrix (out_mat_28);
main.c:45:9: missed: statement clobbers memory: printf ("Error: not enough memory to run the test using n = %i\n",
param_n.3_8);
main.c:28:9: missed: statement clobbers memory: printf ("Usage: %s <n> \n", _1);
main.c:29:9: missed: statement clobbers memory: __builtin_puts (&"  <n> is the desired test size."[0]);
main.c:24:5: note: ***** Analysis failed with vector mode V4SI
main.c:24:5: note: ***** Skipping vector mode V16QI, which would repeat the analysis for V4SI
```

This reveals that the optimization capabilities of *gcc* and *nvc* are significantly different. This indicates that Codee is a good complement to the compiler and enables the programmer to create performant code for the GPU.

The runtime of the MATMUL sequential version written in C and compiled with *gcc* maximum optimization level (*-fast*) is 33.4 seconds because *gcc* does not apply loop interchange automatically. It goes down to 8.8 seconds using the version with explicit loop interchange.

```
$ gcc -Ofast clock.c matrix.c main.c -I include/ -o matmul

$ ./matmul 3000
- Input parameters
n       = 3000
- Executing test...
time (s)= 33.369070
size    = 3000
chksum  = 546933812237

$ gcc -Ofast clock.c matrix.c main_li.c -I include/ -o matmul_li

$ ./matmul_li 3000
- Input parameters
n       = 3000
- Executing test...
time (s)= 8.808224
size    = 3000
chksum  = 546933812237
```

Finally, regarding the OpenACC-enabled version, the *gcc* compiler does not provide OpenACC support mature enough to generate efficient code for the GPU [9].

```
$ module use /global/cfs/cdirs/m1759/yunhe/Modules/perlmutter/modulefiles
$ module load gcc/12.1.0


$ gcc -Ofast -openacc -foffload=nvptx-none="-Ofast -misa=sm_80" clock.c matrix.c main_li_acc.c -I include/ -o
matmul_li_acc

$ ./matmul_li_acc 3000
- Input parameters
n         = 3000
- Executing test...
time (s)= 8.828352
size      = 3000
chksum   = 546933812237


$ gcc -Ofast -fopenmp -foffload=nvptx-none="-Ofast -misa=sm_80" matrix.c clock.c main_li_omp.c -o matmul_li_omp -I
include

$ ./matmul_li_omp 3000
- Input parameters
n     = 3000
- Executing test...
libgomp: cuCtxSynchronize error: an illegal memory access was encountered
libgomp: cuMemFree_v2 error: an illegal memory access was encountered
libgomp: device finalization failed
```

[9] Note the *gcc 12.1* compiler combined Codee 2023.1 raises a runtime error under investigation.