codee

Quickstart Guide of Codee
Matrix-Matrix Multiplication (MATMUL-Fortran)
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 Fortran programming language considered in this case is shown below [3]. Note it is a standard implementation using 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 *nvfortran* 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_april_2023/* once uncompressed.

```
 1   SUBROUTINE calculate_matmul(N,A,B,C)
 2       real*8, dimension(1:N,1:N) :: A, B, C
 3       INTEGER :: i, j, k , N
 4       ! Initialization
 5       DO i = 1, N
 6           DO j = 1, N
 7               C(i,j) = 0.0
 8           END DO
 9       END DO
10       ! Accumulation
11       DO i = 1, N
12           DO j = 1, N
13               DO k = 1, N
14                   C(i,j) = C(i,j) + ( A(i,k) * B(k,j) )
15               END DO
16           END DO
17       END DO
18   END
```

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 (*matmul.f90*) and its corresponding compilation flags after the special mark "*--*" (*-fast*). In this case, the screening reported 14 checks in the source code, the breakdown being 7 checks related to memory issues, 3 checks related to vectorization, 2 checks related to multithreading and 2 checks related to offloading.

```
$ pwreport --screening matmul.f90 --include-tags all -- -fast
Compiler flags: -fast

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

pwreport: warning: Fortran support is experimental
SCREENING REPORT

Target     Lines of code Optimizable lines Analysis time # checks Effort Cost    Profiling
---------- ------------- ----------------- ------------- -------- ------ ------- ---------
matmul.f90 81            29                282 ms        14       108 h  3534€   n/a
---------- ------------- ----------------- ------------- -------- ------ ------- ---------
Total      81            29                282 ms        14       108 h  3534€   n/a

CHECKS PER STAGE OF THE PERFORMANCE OPTIMIZATION ROADMAP
Target     Scalar Control Memory Vector Multi Offload
---------- ------ ------- ------ ------ ----- -------
matmul.f90 0      0       7      3      2     2
---------- ------ ------- ------ ------ ----- -------
Total      0      0       7      3      2     2

TOTAL NUMBER OF LOOPS FOR CODEE AUTO AND GUIDED MODES
                 ---------------- # checks ----------------
Codee mode # Loops Scalar Control Memory Vector Multi Offload
---------- ------- ------ ------- ------ ------ ----- -------
Auto       0       0      0       0      0      0     0
Guided     6       0      0       3      3      2     2

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 matmul.f90 --include-tags all -- -fast

  You can specify multiple inputs which will be displayed as multiple rows (ie. targets) in the table, eg:
        pwreport --screening some/other/dir matmul.f90 --include-tags all -- -fast

  Use --checks to find out details about the detected checks:
        pwreport --checks matmul.f90 --include-tags all -- -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 matmul.f90 -- -fast

1 file successfully analyzed and 0 failures in 362 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 matmul.f90:calculate_matmul --include-tags all -- -fast
Compiler flags: -fast

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

pwreport: warning: Fortran support is experimental
CHECKS REPORT

matmul.f90:1:1 [PWR007]: procedure 'calculate_matmul' does not have or inherit IMPLICIT NONE policy
matmul.f90:1:1 [PWR008]: procedure 'calculate_matmul' has parameters missing INTENT: 'N', 'A', 'B' and 'C'
matmul.f90:5:5 [PWR039]: consider loop interchange to improve the locality of reference and enable vectorization
matmul.f90:6:9 [RMK010]: the vectorization cost model states the loop is not a SIMD opportunity due to strided memory
accesses in the loop body
matmul.f90:11:5 [PWR035]: avoid non-consecutive array access for variables 'A' and 'C' to improve performance
matmul.f90:11:5 [PWR050]: consider applying multithreading parallelism to forall loop
matmul.f90:11:5 [PWR055]: consider applying offloading parallelism to forall loop
matmul.f90:13:13 [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 matmul.f90:calculate_matmul --include-tags all -- -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 101 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 (PWR035) [4] and a loop that is a candidate to be offloaded to the GPU (PWR055).

```
$ pwreport --verbose --checks matmul.f90:calculate_matmul --include-tags all -- -fast
Compiler flags: -fast

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

pwreport: warning: Fortran support is experimental
CHECKS REPORT

matmul.f90:1:1 [PWR007]: procedure 'calculate_matmul' does not have or inherit IMPLICIT NONE policy
  Suggestion: add IMPLICIT NONE in the specification part of the procedure
  Documentation: https://www.codee.com/knowledge/pwr007

matmul.f90:1:1 [PWR008]: procedure 'calculate_matmul' has parameters missing INTENT: 'N', 'A', 'B' and 'C'
  Suggestion: add the appropriate INTENT for each parameter
  Documentation: https://www.codee.com/knowledge/pwr008

matmul.f90:5:5 [PWR039]: consider loop interchange to improve the locality of reference and enable vectorization
  Loops to interchange:
    5:     DO i = 1, N
    6:         DO j = 1, N
  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 matmul.f90:5:5 -- -fast

matmul.f90:6:9 [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

matmul.f90:11:5 [PWR035]: avoid non-consecutive array access for variables 'A' and 'C' to improve performance
  Non-consecutive array access:
    14:                 C(i,j) = 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' and 'C'.
  Documentation: https://www.codee.com/knowledge/pwr035

matmul.f90:11:5 [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 matmul.f90:11:5 -- -fast
      * Using OpenMP 'taskwait':
        pwdirectives --multi omp-taskwait --in-place matmul.f90:11:5 -- -fast
      * Using OpenMP 'taskloop':
        pwdirectives --multi omp-taskloop --in-place matmul.f90:11:5 -- -fast

matmul.f90:11:5 [PWR055]: consider applying offloading parallelism to forall loop
  Suggestion: use pwdirectives to automatically optimize the code
  Documentation: https://www.codee.com/knowledge/pwr055
  AutoFix (choose one option):
    * Using OpenMP (recommended):
        pwdirectives --offload omp-teams --in-place matmul.f90:11:5 -- -fast
    * Using OpenAcc:
        pwdirectives --offload acc --in-place matmul.f90:11:5 -- -fast

matmul.f90:13:13 [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/
```

[4] Note that the support of this Fortran code in PWR039 (loop interchange) is work in progress.

```
1 file successfully analyzed and 0 failures in 83 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 checks reported by Codee that point out memory inefficiencies. In this case, Codee reports memory inefficiencies due to non-consecutive memory accesses (PWR035). As a result, single-core optimizations consist of enforcing loop interchange to enable sequential memory accesses.

For illustrative purposes, some Fortran compilers are capable of applying loop interchange to fix memory inefficiencies automatically, without the intervention of the programmer. The *nvfortran* command below reports that loop interchange has been automatically applied to two loop nests inside the *calculate_matmul* function.

```
$ nvfortran -fast matmul.f90 -o matmul -Minfo=all
calculate_matmul:
      5, Loop interchange produces reordered loop nest: 6,5
         Generated vector simd code for the loop
      6, Zero trip check eliminated
         Loop not fused: different controlling conditions
     11, Loop interchange produces reordered loop nest: 12,13,11
         Generated vector simd code for the loop
     12, Zero trip check eliminated
     13, Zero trip check eliminated
     14, FMA (fused multiply-add) instruction(s) generated
checksum:
     26, Loop interchange produces reordered loop nest: 27,26
         Generated vector simd code for the loop containing reductions
     27, Zero trip check eliminated
matmul_main:
     63, Loop interchange produces reordered loop nest: 64,63
         Generated vector simd code for the loop
     64, Zero trip check eliminated
         Loop not fused: function call before adjacent loop
     73, Loop not vectorized/parallelized: contains call
     94, Memory copy idiom, array assignment replaced by call to pgf90_mcopy8
```

Consider the new source code below (file named "*matmul_li.f90*") that explicitly enforces loop interchange in these loop nests [5]:

[5] Note Codee 2023.1 does not generate this Fortran source code with loop interchange, it is work in progress.

```
 1   SUBROUTINE calculate_matmul(N,A,B,C)
 2       real*8, dimension(1:N,1:N) :: A, B, C
 3       INTEGER :: i, j, k, N
 4       ! Initialization
 5       DO j = 1, N
 6           DO i = 1, N
 7               C(i,j) = 0.0
 8           END DO
 9       END DO
10       ! Accumulation
11       DO j = 1, N
12           DO k = 1, N
13               DO i = 1, N
14                   C(i,j) = C(i,j) + ( A(i,k) * B(k,j) )
15               END DO
16           END DO
17       END DO
18   END
```

Finally, let's benchmark the original source code ("*matmul.f90*") and the optimized version based on loop interchange ("*matmul_li.f90*"). As shown below, the single-core performance gain is ~1x using the Nvidia *nvfortran* compiler with maximum optimization level (*-fast*), as *nvfortran* successfully applies loop interchange automatically to remove the memory inefficiency (the runtime is 13.2 and 13.9 seconds, respectively).

```
$ nvfortran -fast matmul.f90 -o matmul
$ ./matmul 3000
 time (s) =     13.20597839355469
 size     =          3000
 Chksum    =    5.4681062849325534E+023

$ nvfortran -fast matmul_li.f90 -o matmul_li
$ ./matmul_li
 time (s) =     13.91109180450439
 size     =          3000
 Chksum    =    5.4681062849325534E+023
```

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 "*matmul_li.f90*"), which points out an PWR055:

```
...

matmul.f90:11:5 [PWR055]: consider applying offloading parallelism to forall loop
  Suggestion: use pwdirectives to automatically optimize the code
  Documentation: https://www.codee.com/knowledge/pwr055
  AutoFix (choose one option):
    * Using OpenMP (recommended):
        pwdirectives --offload omp-teams --in-place matmul.f90:11:5 -- -fast
    * Using OpenAcc:
        pwdirectives --offload acc --in-place matmul.f90:11:5 -- -fast

...
```

First, produce a further optimized source code (new file name "*matmul_li_acc.f90*") 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 [6]:

```
$ pwdirectives --offload acc -o matmul_li_acc.f90 matmul_li.f90:11:5 -- -fast
Compiler flags: -fast

pwdirectives: warning: Fortran support is experimental
[Fortran] target compiler: <none> (Compiler Agnostic Mode)

Results for file 'matmul_li.f90':
  Successfully parallelized loop at 'matmul_li.f90:calculate_matmul:11:5' [using offloading without teams]:
      [INFO] matmul_li.f90:11:5 Parallel forall: variable 'C'
      [INFO] matmul_li.f90:11:5 Parallel region defined by OpenACC directive 'parallel'
      [INFO] matmul_li.f90:11:5 Loop parallelized with OpenACC directive 'loop'
      [INFO] matmul_li.f90:11:5 Complete access range for variables: 'A', 'B', 'C'
      [INFO] matmul_li.f90:11:5 Data region for host-device data transfers defined by OpenACC directive 'data'
Successfully created matmul_li_acc.f90

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

Thus, the resulting MATMUL version shown below implements loop interchange and OpenACC offload. Note the clauses "*copyin(N, A(:), B(:)) copy(C(:))*" do not specify the ranges (line 11), which must be filled in explicitly by the programmer as "*copyin(N, A(1:N,1:N), B(1:N,1:N)) copy(C(1:N,1:N))*". The resulting source code is as follows:

```
 1   SUBROUTINE calculate_matmul(N,A,B,C)
 2       real*8, dimension(1:N,1:N) :: A, B, C
 3       INTEGER :: i, j, k, N
 4       ! Initialization
 5       DO j = 1, N
 6           DO i = 1, N
 7               C(i,j) = 0.0
 8           END DO
 9       END DO
10       ! Accumulation
11       !$acc data copyin(N, A(1:N,1:N), B(1:N,1:N)) copy(C(1:N,1:N))
12       !$acc parallel
13       !$acc loop
14       DO j = 1, N
15           DO k = 1, N
16               DO i = 1, N
17                   C(i,j) = C(i,j) + ( A(i,k) * B(k,j) )
18               END DO
19           END DO
20       END DO
21       !$acc end parallel
22       !$acc end data
23   END
```

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

```
$ pwdirectives --offload omp-teams -o matmul_li_omp.f90 matmul_li.f90:11:5 -- -fast
Compiler flags: -fast

pwdirectives: warning: Fortran support is experimental
[Fortran] target compiler: <none> (Compiler Agnostic Mode)
```

[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 matmul_li_acc.f90*".

```
Results for file 'matmul_li.f90':
  Successfully parallelized loop at 'matmul_li.f90:calculate_matmul:11:5' [using offloading]:
      [INFO] matmul_li.f90:11:5 Parallel forall: variable 'C'
      [INFO] matmul_li.f90:11:5 Complete access range for variables: 'A', 'B', 'C'
      [INFO] matmul_li.f90:11:5 Loop parallelized with teams using OpenMP directive 'target teams distribute parallel
for'
Successfully created matmul_li_omp.f90

Minimum software stack requirements: OpenMP version 4.0 with offloading capabilities
```

Thus, the resulting MATMUL version shown below implements loop interchange and OpenMP offload. Note the clauses "*map(to: N, A(:), B(:)) map(tofrom: C(:))*" do not specify the ranges (line 11), which must be filled in explicitly by the programmer as "*map(to: N, A(1:N,1:N), B(1:N,1:N)) map(tofrom: C(1:N,1:N))*". The resulting source code is as follows:

```
     1   SUBROUTINE calculate_matmul(N,A,B,C)
     2       real*8, dimension(1:N,1:N) :: A, B, C
     3       INTEGER :: i, j, k, N
     4       ! Initialization
     5       DO j = 1, N
     6           DO i = 1, N
     7               C(i,j) = 0.0
     8           END DO
     9       END DO
    10       ! Accumulation
    11        !$omp target teams distribute parallel do shared(A, B, N) map(to: N, A(1:N,1:N), B(1:N,1:N)) private(i,
k) map(tofrom: C(1:N,1:N)) schedule(static)
    12       DO j = 1, N
    13           DO k = 1, N
    14               DO i = 1, N
    15                   C(i,j) = C(i,j) + ( A(i,k) * B(k,j) )
    16               END DO
    17           END DO
    18       END DO
    19   END
```

Finally, let's benchmark the original source code ("*matmul.f90*") versus the optimized version based on loop interchange and GPU offload directives, both OpenACC offload ("*matmul_li_acc.f90*") and OpenMP offload ("*matmul_li_omp.f90*"). As shown below, the GPU-enabled performance gain is 24.1x using the Nvidia *nvfortran* compiler with maximum optimization level (*-fast*), reducing the runtime from 12.5 seconds down to 0.5 seconds with OpenACC. The OpenMP version compiles successfully, failing at runtime with a fatal error [7] [8].

```
$ nvfortran -fast matmul.f90 -o matmul
$ ./matmul 3000
 time (s) =     13.20597839355469
 size     =          3000
 Chksum    =    5.4681062849325534E+023

$ nvfortran -fast -acc -target=gpu -Minfo=acc matmul_li_acc.f90 -o matmul_li_acc
```

[7] Note about *nvfortran* support for OpenMP offload: Codee produces OpenMP offload code that uses the clause "*schedule(static)*". If this clause is removed, then *nvfortran* produces correct code but with poor efficiency. For instance, a typical example run is as follows (poor runtime 16.5 seconds compared with 1.1 seconds of the OpenACC version):

```
$ ./matmul_li_omp
 time (s) =    16.56595039367676
 size     =         3000
 Chksum    =    5.4681062849325534E+023
```

[8] Note about *nvfortran* support for Openmp offload: Alternatively the pragma "*!$omp target teams distribute parallel do*" can be replaced by "*$omp target teams loop*", and then *nvfortran* produces correct code that runs in 1.2 seconds (compared to OpenACC version that runs in 0.5 seconds).

```
$ ./matmul_li_acc 3000
 time (s) =    0.5208020210266113
 size     =          3000
 Chksum    =    5.4681062849325534E+023

$ nvfortran -fast -mp -target=gpu -Minfo=mp matmul_li_omp.f90 -o matmul_li_omp
$ ./matmul_li_omp 3000
Fatal error: expression 'HX_CU_CALL_CHECK(p_cuStreamSynchronize(stream[dev]))' (value 1) is not equal to expression
'HX_SUCCESS' (value 0)
Aborted
```

Step 7. Submitting a job to a GPU node

Pelmutter has at the disposal of the user 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 [9]:

```
#!/bin/bash
#SBATCH -A ntrain8
#SBATCH --reservation=codee_day1_gpu
#SBATCH -C gpu
#SBATCH -J Codee_Matmul_Fortran
#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_FORTRAN.sh
```

Finally, the script *MATMUL_FORTRAN.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

echo ""
echo ""
```

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

```
echo "Matmul"
printRunCommand "nvfortran -fast matmul.f90 -o matmul"
printRunCommand "./matmul 3000"

echo ""
echo ""
echo "Matmul with loop interchange"
printRunCommand "nvfortran -fast matmul_li.f90 -o matmul_li"
printRunCommand "./matmul_li 3000"

echo ""
echo ""
echo "Matmul with loop interchange and OpenACC"
printRunCommand "nvfortran -fast -acc -target=gpu -Minfo=acc matmul_li_acc.f90 -o matmul_li_acc"
printRunCommand "./matmul_li_acc 3000"

echo ""
echo ""
echo "Matmul with loop interchange and OpenMP"
printRunCommand "nvfortran -fast -mp -target=gpu -Minfo=mp matmul_li_omp.f90 -o matmul_li_omp"
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 [10]. 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 *nvfortran* compiler: (1) applies loop interchange to the Fortran source code in the two loop nests of the *calculate_matmul* function, and (2) reports attempts to vectorize loops but does not report opportunities in loops that might be offloaded to the GPU.

```
$ nvfortran -fast matmul.f90 -o matmul -Minfo=all
calculate_matmul:
      5, Loop interchange produces reordered loop nest: 6,5
         Generated vector simd code for the loop
      6, Zero trip check eliminated
         Loop not fused: different controlling conditions
     11, Loop interchange produces reordered loop nest: 12,13,11
         Generated vector simd code for the loop
     12, Zero trip check eliminated
     13, Zero trip check eliminated
     14, FMA (fused multiply-add) instruction(s) generated
checksum:
     26, Loop interchange produces reordered loop nest: 27,26
         Generated vector simd code for the loop containing reductions
     27, Zero trip check eliminated
matmul_main:
     63, Loop interchange produces reordered loop nest: 64,63
         Generated vector simd code for the loop
     64, Zero trip check eliminated
         Loop not fused: function call before adjacent loop
     73, Loop not vectorized/parallelized: contains call
     94, Memory copy idiom, array assignment replaced by call to pgf90_mcopy8
```

Regarding the OpenACC-enabled source code with loop interchange explicitly coded in Fortran, the user messages reported by the *nvfortran* 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.

```
$ nvfortran -fast -acc -target=gpu -Minfo=all matmul_li_acc.f90 -o matmul_acc
calculate_matmul:
      6, Memory zero idiom, loop replaced by call to __c_mzero8
     11, Generating copyin(n) [if not already present]
         Generating copy(c(:n,:n)) [if not already present]
         Generating copyin(a(:n,:n),b(:n,:n)) [if not already present]
     12, Generating NVIDIA GPU code
         14, !$acc loop gang ! blockidx%x
         15, !$acc loop seq
         16, !$acc loop vector(128) ! threadidx%x
     15, Loop carried dependence of c prevents parallelization
         Loop carried backward dependence of c prevents vectorization
     16, Loop is parallelizable
```

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

```
checksum:
     31, Loop interchange produces reordered loop nest: 32,31
         Generated vector simd code for the loop containing reductions
     32, Zero trip check eliminated
matmul_main:
     68, Loop interchange produces reordered loop nest: 69,68
         Generated vector simd code for the loop
     69, Zero trip check eliminated
         Loop not fused: function call before adjacent loop
     78, Loop not vectorized/parallelized: contains call
     99, Memory copy idiom, array assignment replaced by call to pgf90_mcopy8
/usr/bin/ld: warning: /tmp/pgcudafatd_GljZlgzzZe.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
```

```
$ nvfortran -fast -mp -target=gpu -Minfo=all matmul_li_omp.f90 -o matmul_li_omp
calculate_matmul:
      6, Memory zero idiom, loop replaced by call to __c_mzero8
     11, !$omp target teams distribute parallel do
         11, Generating "nvkernel_calculate_matmul__F1L11_2" GPU kernel
     11, Generating map(to:n)
         Generating map(tofrom:c(:n,:n))
         Generating map(to:a(:n,:n),b(:n,:n))
checksum:
     27, Loop interchange produces reordered loop nest: 28,27
         Generated vector simd code for the loop containing reductions
     28, Zero trip check eliminated
matmul_main:
     64, Loop interchange produces reordered loop nest: 65,64
         Generated vector simd code for the loop
     65, Zero trip check eliminated
         Loop not fused: function call before adjacent loop
     74, Loop not vectorized/parallelized: contains call
     95, Memory copy idiom, array assignment replaced by call to pgf90_mcopy8
/usr/bin/ld: warning: /tmp/pgcudafatcSklgLzFqVSV.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 MATMUL sequential source code, it is noticeable that the GNU *gfortran* compiler: (1) applies loop interchange to the Fortran source code in the first loop nest of the *calculate_matmul* function, (2) it does not apply loop interchange in the second loop nest, and (3) reports attempts to vectorize loops but does not report opportunities in loops that might be offloaded to the GPU. For illustrative purposes, see the messages reported by *gfortran* set up in maximum optimization level (*-fast*) and in verbose mode (*-fopt-info-all*):

```
$ module load PrgEnv-gnu

$ gfortran matmul.f90 -fopt-info-all -Ofast -o matmul
matmul.f90:84:26: note: Considering inline candidate checksum/0.
matmul.f90:84:26: optimized:  Inlining checksum/0 into matmul_main/2.
matmul.f90:74:41: note: Considering inline candidate calculate_matmul/1.
matmul.f90:74:41: missed:    will not early inline: matmul_main/2->calculate_matmul/1, call is cold and code would grow
at least by 30
matmul.f90:104:3: note: Considering inline candidate matmul_main/2.
matmul.f90:104:3: missed:   not inlinable: main/3 -> matmul_main/2, --param large-stack-frame-growth limit reached
matmul.f90:104:3: missed:   not inlinable: main/3 -> matmul_main/2, --param max-inline-insns-auto limit reached
matmul.f90:104:3: missed:   not inlinable: main/3 -> _gfortran_set_options/5, function body not available
matmul.f90:104:3: missed:   not inlinable: main/3 -> _gfortran_set_args/4, function body not available
matmul.f90:102:17: missed:   not inlinable: matmul_main/2 -> __builtin_free/22, function body not available
matmul.f90:102:15:  missed:      not  inlinable:  matmul_main/2  ->  _gfortran_runtime_error_at/13,  function  body  not
available
matmul.f90:101:17: missed:   not inlinable: matmul_main/2 -> __builtin_free/22, function body not available
matmul.f90:101:15:  missed:      not  inlinable:  matmul_main/2  ->  _gfortran_runtime_error_at/13,  function  body  not
available
matmul.f90:100:17: missed:   not inlinable: matmul_main/2 -> __builtin_free/22, function body not available
matmul.f90:100:15:  missed:      not  inlinable:  matmul_main/2  ->  _gfortran_runtime_error_at/13,  function  body  not
available
matmul.f90:90:29: missed:   not inlinable: matmul_main/2 -> _gfortran_st_write_done/20, function body not available
matmul.f90:90:29:  missed:     not  inlinable:  matmul_main/2  ->  _gfortran_transfer_real_write/19,  function  body  not
available
matmul.f90:90:25:  missed:    not inlinable: matmul_main/2 -> _gfortran_transfer_character_write/18, function body not
available
matmul.f90:90:29: missed:   not inlinable: matmul_main/2 -> _gfortran_st_write/17, function body not available
matmul.f90:86:28: missed:   not inlinable: matmul_main/2 -> _gfortran_st_write_done/20, function body not available
matmul.f90:86:28:  missed:    not  inlinable:  matmul_main/2  ->  _gfortran_transfer_integer_write/21,  function  body  not
available
matmul.f90:86:25:  missed:    not inlinable: matmul_main/2 -> _gfortran_transfer_character_write/18, function body not
available
matmul.f90:86:28: missed:   not inlinable: matmul_main/2 -> _gfortran_st_write/17, function body not available
matmul.f90:85:39: missed:   not inlinable: matmul_main/2 -> _gfortran_st_write_done/20, function body not available
matmul.f90:85:39:  missed:     not  inlinable:  matmul_main/2  ->  _gfortran_transfer_real_write/19,  function  body  not
available
matmul.f90:85:25:  missed:    not inlinable: matmul_main/2 -> _gfortran_transfer_character_write/18, function body not
available
matmul.f90:85:39: missed:   not inlinable: matmul_main/2 -> _gfortran_st_write/17, function body not available
matmul.f90:76:43: missed:   not inlinable: matmul_main/2 -> _gfortran_system_clock_4/16, function body not available
matmul.f90:74:41:  missed:    not  inlinable:  matmul_main/2  ->  calculate_matmul/1,  --param  max-inline-insns-auto  limit
reached
matmul.f90:72:45: missed:   not inlinable: matmul_main/2 -> _gfortran_system_clock_4/16, function body not available
matmul.f90:71:70: missed:   not inlinable: matmul_main/2 -> _gfortran_system_clock_4/16, function body not available
matmul.f90:60:21: missed:   not inlinable: matmul_main/2 -> _gfortran_os_error_at/15, function body not available
matmul.f90:60:21: missed:   not inlinable: matmul_main/2 -> __builtin_malloc/14, function body not available
matmul.f90:60:21: missed:   not inlinable: matmul_main/2 -> _gfortran_runtime_error/12, function body not available
matmul.f90:59:21: missed:   not inlinable: matmul_main/2 -> _gfortran_os_error_at/15, function body not available
matmul.f90:59:21: missed:   not inlinable: matmul_main/2 -> __builtin_malloc/14, function body not available
matmul.f90:59:21: missed:   not inlinable: matmul_main/2 -> _gfortran_runtime_error/12, function body not available
matmul.f90:58:21: missed:   not inlinable: matmul_main/2 -> _gfortran_os_error_at/15, function body not available
matmul.f90:58:21: missed:   not inlinable: matmul_main/2 -> __builtin_malloc/14, function body not available
matmul.f90:58:21: missed:   not inlinable: matmul_main/2 -> _gfortran_runtime_error/12, function body not available
matmul.f90:51:18: missed:   not inlinable: matmul_main/2 -> _gfortran_st_read_done/11, function body not available
matmul.f90:51:18: missed:   not inlinable: matmul_main/2 -> _gfortran_transfer_integer/10, function body not available
matmul.f90:51:18: missed:   not inlinable: matmul_main/2 -> _gfortran_st_read/9, function body not available
matmul.f90:49:38:  missed:    not  inlinable:  matmul_main/2  ->  _gfortran_get_command_argument_i4/8,  function  body  not
available
matmul.f90:47:8: missed:   not inlinable: matmul_main/2 -> _gfortran_iargc/7, function body not available
Unit growth for small function inlining: 306->306 (0%)
matmul.f90:104:3: missed:   not inlinable: main/3 -> matmul_main/2, --param large-stack-frame-growth limit reached
```

```
matmul.f90:104:3: missed:   not inlinable: main/3 -> matmul_main/2, --param large-stack-frame-growth limit reached

Inlined 1 calls, eliminated 0 functions

matmul.f90:26:9: missed: couldn't vectorize loop
matmul.f90:28:30: missed: not vectorized: not suitable for strided load _8 = (*c_21(D))[_7];
matmul.f90:27:13: missed: couldn't vectorize loop
matmul.f90:28:30: missed: not vectorized: no vectype for stmt: _8 = (*c_21(D))[_7];
 scalar_type: real(kind=8)
matmul.f90:20:19: note: vectorized 0 loops in function.
matmul.f90:31:3: note: ***** Analysis failed with vector mode V4SI
matmul.f90:31:3: note: ***** Skipping vector mode V16QI, which would repeat the analysis for V4SI
matmul.f90:11:10: missed: couldn't vectorize loop
matmul.f90:11:10: missed: not vectorized: multiple nested loops.
matmul.f90:12:14: missed: couldn't vectorize loop
matmul.f90:14:53: missed: not vectorized: not suitable for strided load _17 = (*a_41(D))[_16];
matmul.f90:13:18: optimized: loop vectorized using 16 byte vectors
matmul.f90:5:10: missed: couldn't vectorize loop
matmul.f90:7:24: missed: not vectorized: not suitable for strided load (*c_37(D))[_6] = 0.0;
matmul.f90:6:14: missed: couldn't vectorize loop
matmul.f90:7:24: missed: not vectorized: no vectype for stmt: (*c_37(D))[_6] = 0.0;
 scalar_type: real(kind=8)
matmul.f90:1:27: note: vectorized 1 loops in function.
matmul.f90:13:18: optimized: loop turned into non-loop; it never loops
matmul.f90:14:53: note: ***** Analysis failed with vector mode V4SI
matmul.f90:14:53: note: ***** The result for vector mode V16QI would be the same
matmul.f90:14:53: note: ***** Re-trying analysis with vector mode V8QI
matmul.f90:14:53: note: ***** Analysis failed with vector mode V8QI
matmul.f90:73:14: optimized: loop with 2 iterations completely unrolled (header execution count 14598063)
matmul.f90:49:38: missed: statement clobbers memory: _gfortran_get_command_argument_i4 (&C.4058, &num, 0B, 0B, 100);
matmul.f90:26:9: missed: couldn't vectorize loop
matmul.f90:28:30:  missed:  not  vectorized:  not  suitable  for  strided  load  _172  =  MEM  <real(kind=8)[0:D.4485]>
[(real(kind=8)[0:D.3949] *)_37][_171];
matmul.f90:27:13: missed: couldn't vectorize loop
matmul.f90:28:30:   missed:   not   vectorized:   no   vectype   for   stmt:   _172   =   MEM   <real(kind=8)[0:D.4485]>
[(real(kind=8)[0:D.3949] *)_37][_171];
 scalar_type: real(kind=8)
matmul.f90:63:9: missed: couldn't vectorize loop
matmul.f90:65:35:  missed:  not  vectorized:  not  suitable  for  strided  load MEM <real(kind=8)[0:]> [(real(kind=8)[0:]
*)_14][_50] = _51;
matmul.f90:64:12: missed: couldn't vectorize loop
matmul.f90:65:35: missed: not vectorized: no vectype for stmt: MEM <real(kind=8)[0:]> [(real(kind=8)[0:] *)_14][_50] =
_51;
 scalar_type: real(kind=8)
matmul.f90:33:19: note: vectorized 0 loops in function.
matmul.f90:47:8: missed: statement clobbers memory: _1 = _gfortran_iargc ();
matmul.f90:49:38: missed: statement clobbers memory: _gfortran_get_command_argument_i4 (&C.4058, &num, 0B, 0B, 100);
matmul.f90:51:18: missed: statement clobbers memory: _gfortran_st_read (&dt_parm.20);
matmul.f90:51:18: missed: statement clobbers memory: _gfortran_transfer_integer (&dt_parm.20, &n, 4);
matmul.f90:51:18: missed: statement clobbers memory: _gfortran_st_read_done (&dt_parm.20);
matmul.f90:58:21: missed: statement clobbers memory: _14 = __builtin_malloc (_13);
matmul.f90:59:21: missed: statement clobbers memory: _26 = __builtin_malloc (_13);
matmul.f90:60:21: missed: statement clobbers memory: _37 = __builtin_malloc (_13);
matmul.f90:71:70: missed: statement clobbers memory: _gfortran_system_clock_4 (0B, &count_rate.30, &count_max.31);
matmul.f90:72:45: missed: statement clobbers memory: _gfortran_system_clock_4 (&count.32, 0B, 0B);
matmul.f90:74:41: missed: statement clobbers memory: calculate_matmul (&n, _14, _26, _37);
matmul.f90:74:41: missed: statement clobbers memory: calculate_matmul (&n, _14, _26, _37);
matmul.f90:76:43: missed: statement clobbers memory: _gfortran_system_clock_4 (&count.33, 0B, 0B);
matmul.f90:85:39: missed: statement clobbers memory: _gfortran_st_write (&dt_parm.34);
matmul.f90:85:25:  missed:  statement  clobbers  memory:  _gfortran_transfer_character_write  (&dt_parm.34,  &"time  (s)  =
"[1]{lb: 1 sz: 1}, 11);
matmul.f90:85:39: missed: statement clobbers memory: _gfortran_transfer_real_write (&dt_parm.34, &elapsed_time, 8);
matmul.f90:85:39: missed: statement clobbers memory: _gfortran_st_write_done (&dt_parm.34);
matmul.f90:86:28: missed: statement clobbers memory: _gfortran_st_write (&dt_parm.35);
matmul.f90:86:25:  missed:  statement  clobbers  memory:  _gfortran_transfer_character_write  (&dt_parm.35,  &"size      =
"[1]{lb: 1 sz: 1}, 11);
matmul.f90:86:28: missed: statement clobbers memory: _gfortran_transfer_integer_write (&dt_parm.35, &n, 4);
matmul.f90:86:28: missed: statement clobbers memory: _gfortran_st_write_done (&dt_parm.35);
matmul.f90:90:29: missed: statement clobbers memory: _gfortran_st_write (&dt_parm.37);
matmul.f90:90:25:  missed:  statement  clobbers  memory:  _gfortran_transfer_character_write  (&dt_parm.37,  &"Chksum     =
"[1]{lb: 1 sz: 1}, 12);
matmul.f90:90:29: missed: statement clobbers memory: _gfortran_transfer_real_write (&dt_parm.37, &chk, 8);
matmul.f90:90:29: missed: statement clobbers memory: _gfortran_st_write_done (&dt_parm.37);
matmul.f90:100:17: missed: statement clobbers memory: __builtin_free (_14);
matmul.f90:101:17: missed: statement clobbers memory: __builtin_free (_26);
matmul.f90:102:17: missed: statement clobbers memory: __builtin_free (_37);
matmul.f90:60:21:  missed:  statement  clobbers  memory:  _gfortran_os_error_at  (&"In  file  \'matmul.f90\',  around  line
```

```
61"[1]{lb: 1 sz: 1}, &"Error allocating %lu bytes"[1]{lb: 1 sz: 1}, size.21_164);
matmul.f90:59:21: missed: statement clobbers memory: _gfortran_os_error_at (&"In file \'matmul.f90\', around line
60"[1]{lb: 1 sz: 1}, &"Error allocating %lu bytes"[1]{lb: 1 sz: 1}, size.21_164);
matmul.f90:58:21: missed: statement clobbers memory: _gfortran_os_error_at (&"In file \'matmul.f90\', around line
59"[1]{lb: 1 sz: 1}, &"Error allocating %lu bytes"[1]{lb: 1 sz: 1}, size.21_164);
matmul.f90:58:21: missed: statement clobbers memory: _gfortran_runtime_error (&"Integer overflow when calculating the
amount of memory to allocate"[1]{lb: 1 sz: 1});
matmul.f90:90:29: note: ***** Analysis succeeded with vector mode V4SI
matmul.f90:90:29: note: SLPing BB part
matmul.f90:51:18: note: Costing subgraph:
matmul.f90:51:18: note: node 0x263ed28 (max_nunits=2, refcnt=1)
matmul.f90:51:18: note: op template: dt_parm.20.common.flags = 16512;
matmul.f90:51:18: note:     stmt 0 dt_parm.20.common.flags = 16512;
matmul.f90:51:18: note:     stmt 1 dt_parm.20.common.unit = -1;
matmul.f90:51:18: note:     children 0x263eda0
matmul.f90:51:18: note: node (constant) 0x263eda0 (max_nunits=1, refcnt=1)
matmul.f90:51:18: note:     { 16512, -1 }
matmul.f90:51:18: note: Cost model analysis:
matmul.f90:51:18: note: Cost model analysis for part in loop 0:
  Vector cost: 24
  Scalar cost: 24
matmul.f90:51:18: note: Basic block will be vectorized using SLP
matmul.f90:51:18: note: Vectorizing SLP tree:
matmul.f90:51:18: note: node 0x263ed28 (max_nunits=2, refcnt=1)
matmul.f90:51:18: note: op template: dt_parm.20.common.flags = 16512;
matmul.f90:51:18: note:     stmt 0 dt_parm.20.common.flags = 16512;
matmul.f90:51:18: note:     stmt 1 dt_parm.20.common.unit = -1;
matmul.f90:51:18: note:     children 0x263eda0
matmul.f90:51:18: note: node (constant) 0x263eda0 (max_nunits=1, refcnt=1)
matmul.f90:51:18: note:     { 16512, -1 }
matmul.f90:51:18: note: ------>vectorizing SLP node starting from: dt_parm.20.common.flags = 16512;
matmul.f90:51:18: note: vect_is_simple_use: operand -1, type of def: constant
matmul.f90:51:18: note: transform store. ncopies = 1
matmul.f90:51:18: note: create vector_type-pointer variable to type: vector(2) integer(kind=4)  vectorizing a pointer
ref: dt_parm.20.common.flags
matmul.f90:51:18: note: created &dt_parm.20.common.flags
matmul.f90:51:18: note:  add new stmt: MEM <vector(2) integer(kind=4)> [(integer(kind=4) *)&dt_parm.20] = { 16512, -1
};
matmul.f90:51:18: note: vectorizing stmts using SLP.
matmul.f90:51:18: optimized: basic block part vectorized using 16 byte vectors
matmul.f90:85:39: note: Costing subgraph:
matmul.f90:85:39: note: node 0x263ee18 (max_nunits=2, refcnt=1)
matmul.f90:85:39: note: op template: dt_parm.34.common.flags = 128;
matmul.f90:85:39: note:     stmt 0 dt_parm.34.common.flags = 128;
matmul.f90:85:39: note:     stmt 1 dt_parm.34.common.unit = 6;
matmul.f90:85:39: note:     children 0x263ee90
matmul.f90:85:39: note: node (constant) 0x263ee90 (max_nunits=1, refcnt=1)
matmul.f90:85:39: note:     { 128, 6 }
matmul.f90:85:39: note: Cost model analysis:
matmul.f90:85:39: note: Cost model analysis for part in loop 0:
  Vector cost: 24
  Scalar cost: 24
matmul.f90:85:39: note: Vectorizing SLP tree:
matmul.f90:85:39: note: node 0x263ee18 (max_nunits=2, refcnt=1)
matmul.f90:85:39: note: op template: dt_parm.34.common.flags = 128;
matmul.f90:85:39: note:     stmt 0 dt_parm.34.common.flags = 128;
matmul.f90:85:39: note:     stmt 1 dt_parm.34.common.unit = 6;
matmul.f90:85:39: note:     children 0x263ee90
matmul.f90:85:39: note: node (constant) 0x263ee90 (max_nunits=1, refcnt=1)
matmul.f90:85:39: note:     { 128, 6 }
matmul.f90:85:39: note: ------>vectorizing SLP node starting from: dt_parm.34.common.flags = 128;
matmul.f90:85:39: note: vect_is_simple_use: operand 6, type of def: constant
matmul.f90:85:39: note: transform store. ncopies = 1
matmul.f90:85:39: note: create vector_type-pointer variable to type: vector(2) integer(kind=4)  vectorizing a pointer
ref: dt_parm.34.common.flags
matmul.f90:85:39: note: created &dt_parm.34.common.flags
matmul.f90:85:39: note: add new stmt: MEM <vector(2) integer(kind=4)> [(integer(kind=4) *)&dt_parm.34] = { 128, 6 };
matmul.f90:85:39: note: vectorizing stmts using SLP.
matmul.f90:85:39: optimized: basic block part vectorized using 16 byte vectors
matmul.f90:86:28: note: Costing subgraph:
matmul.f90:86:28: note: node 0x263ef08 (max_nunits=2, refcnt=1)
matmul.f90:86:28: note: op template: dt_parm.35.common.flags = 128;
matmul.f90:86:28: note:     stmt 0 dt_parm.35.common.flags = 128;
matmul.f90:86:28: note:     stmt 1 dt_parm.35.common.unit = 6;
matmul.f90:86:28: note:     children 0x263ef80
matmul.f90:86:28: note: node (constant) 0x263ef80 (max_nunits=1, refcnt=1)
matmul.f90:86:28: note:     { 128, 6 }
```

```
matmul.f90:86:28: note: Cost model analysis:
matmul.f90:86:28: note: Cost model analysis for part in loop 0:
  Vector cost: 24
  Scalar cost: 24
matmul.f90:86:28: note: Vectorizing SLP tree:
matmul.f90:86:28: note: node 0x263ef08 (max_nunits=2, refcnt=1)
matmul.f90:86:28: note: op template: dt_parm.35.common.flags = 128;
matmul.f90:86:28: note:     stmt 0 dt_parm.35.common.flags = 128;
matmul.f90:86:28: note:     stmt 1 dt_parm.35.common.unit = 6;
matmul.f90:86:28: note:     children 0x263ef80
matmul.f90:86:28: note: node (constant) 0x263ef80 (max_nunits=1, refcnt=1)
matmul.f90:86:28: note:     { 128, 6 }
matmul.f90:86:28: note: ------>vectorizing SLP node starting from: dt_parm.35.common.flags = 128;
matmul.f90:86:28: note: vect_is_simple_use: operand 6, type of def: constant
matmul.f90:86:28: note: transform store. ncopies = 1
matmul.f90:86:28: note: create vector_type-pointer variable to type: vector(2) integer(kind=4)  vectorizing a pointer
ref: dt_parm.35.common.flags
matmul.f90:86:28: note: created &dt_parm.35.common.flags
matmul.f90:86:28: note: add new stmt: MEM <vector(2) integer(kind=4)> [(integer(kind=4) *)&dt_parm.35] = { 128, 6 };
matmul.f90:86:28: note: vectorizing stmts using SLP.
matmul.f90:86:28: optimized: basic block part vectorized using 16 byte vectors
matmul.f90:90:29: note: Costing subgraph:
matmul.f90:90:29: note: node 0x263eff8 (max_nunits=2, refcnt=1)
matmul.f90:90:29: note: op template: dt_parm.37.common.flags = 128;
matmul.f90:90:29: note:     stmt 0 dt_parm.37.common.flags = 128;
matmul.f90:90:29: note:     stmt 1 dt_parm.37.common.unit = 6;
matmul.f90:90:29: note:     children 0x263f070
matmul.f90:90:29: note: node (constant) 0x263f070 (max_nunits=1, refcnt=1)
matmul.f90:90:29: note:     { 128, 6 }
matmul.f90:90:29: note: Cost model analysis:
matmul.f90:90:29: note: Cost model analysis for part in loop 0:
  Vector cost: 24
  Scalar cost: 24
matmul.f90:90:29: note: Vectorizing SLP tree:
matmul.f90:90:29: note: node 0x263eff8 (max_nunits=2, refcnt=1)
matmul.f90:90:29: note: op template: dt_parm.37.common.flags = 128;
matmul.f90:90:29: note:     stmt 0 dt_parm.37.common.flags = 128;
matmul.f90:90:29: note:     stmt 1 dt_parm.37.common.unit = 6;
matmul.f90:90:29: note:     children 0x263f070
matmul.f90:90:29: note: node (constant) 0x263f070 (max_nunits=1, refcnt=1)
matmul.f90:90:29: note:     { 128, 6 }
matmul.f90:90:29: note: ------>vectorizing SLP node starting from: dt_parm.37.common.flags = 128;
matmul.f90:90:29: note: vect_is_simple_use: operand 6, type of def: constant
matmul.f90:90:29: note: transform store. ncopies = 1
matmul.f90:90:29: note: create vector_type-pointer variable to type: vector(2) integer(kind=4)  vectorizing a pointer
ref: dt_parm.37.common.flags
matmul.f90:90:29: note: created &dt_parm.37.common.flags
matmul.f90:90:29: note: add new stmt: MEM <vector(2) integer(kind=4)> [(integer(kind=4) *)&dt_parm.37] = { 128, 6 };
matmul.f90:90:29: note: vectorizing stmts using SLP.
matmul.f90:90:29: optimized: basic block part vectorized using 16 byte vectors
matmul.f90:90:29: note: ***** The result for vector mode V16QI would be the same
matmul.f90:104:3: missed: statement clobbers memory: _gfortran_set_args (argc_2(D), argv_3(D));
matmul.f90:104:3: missed: statement clobbers memory: _gfortran_set_options (7, &options.47[0]);
matmul.f90:104:3: missed: statement clobbers memory: matmul_main ();
matmul.f90:104:3: note: ***** Analysis failed with vector mode VOID
```

This reveals that the optimization capabilities of *gfortran* and *nvfortran* 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 Fortran and compiled with *gfortran* maximum optimization level (*-Ofast*) is 40.5 seconds because *gfortran* does not apply loop interchange automatically. It goes down to 12.2 seconds using the version with explicit loop interchange.

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


$ gfortran -Ofast matmul.f90 -o matmul

$ ./matmul 3000
 time (s) =    40.513999938964844
 size     =          3000
 Chksum    =    5.4681062849326621E+023

$ gfortran -Ofast matmul_li.f90 -o matmul_li

$ ./matmul_li 3000
 time (s) =    12.288000106811523
 size     =          3000
 Chksum    =    5.4681062849326568E+023
```

Finally, regarding the OpenACC-enabled version, the gcc compiler does not provide OpenACC support mature enough to generate efficient code for the GPU [11].

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

$ gfortran -Ofast -openacc -foffload=nvptx-none="-Ofast -misa=sm_80" matmul_li_acc.f90 -o matmul_li_acc

$ ./matmul_li_acc 3000
 time (s) =    12.555999755859375
 size     =          3000
 Chksum    =    5.4681062849326568E+023

$ gfortran -Ofast -fopenmp -foffload=nvptx-none="-Ofast -misa=sm_80" matmul_li_omp.f90 -o matmul_li_omp
matmul_li_omp.f90:11:132:

   11 |     !$omp target teams distribute parallel do shared(A, B, N) map(to: N, A(1:N,1:N), B(1:N,1:N)) private(i, k)
map(tofrom: C(1:N,1:N)) schedule(static)
                                                                                                                    |
1
Error: Line truncated at (1) [-Werror=line-truncation]
matmul_li_omp.f90:11:132:

   11 |     !$omp target teams distribute parallel do shared(A, B, N) map(to: N, A(1:N,1:N), B(1:N,1:N)) private(i, k)
map(tofrom: C(1:N,1:N)) schedule(static)
                                                                                                                    |
1
Error: Invalid form of array reference at (1)
f951: some warnings being treated as errors
```

[11] Note that by default the *PrgEnv-gnu* environment loads *gcc/11.2* and the offload support does not work properly neither with OpenMP or OpenACC. By loading a specific compiler version (in particular *gcc/12.1.0* in the example above in Perlmutter) this error is fixed for OpenACC, but not for OpenMP (in compilation time).