Directive Language Specification

Center for Climate System Modeling (C2SM)
http://www.c2sm.ethz.ch
Valentin Clement

Last updated: December 3, 2018

v2.0

Contents

Conventions used in this document

Keywords and punctuation that are part of the actual specification will appear in typewriter font:

`!$claw`

Italic font is used where a keyword or other name must be used:

`!$claw` *directive-name*

Optional syntax is enclosed in square brackets; an option that may be repeated more than once is followed by ellipses:

`!$claw` *directive-name* [*clause* [[,] *clause*]. . .]

1 General information about the CLAW language

The directives are either local or global.

- Local directive: those directives have a limited impact on a local block of code (for example, only in a subroutine or only in a do statement block)
- Global directive: those directives can have an impact on the whole translation unit.

The CLAW transformations are separated in two sections:

- CLAW high-level abstraction transformations: perform transformations on abstraction based on domain science.
- CLAW low-level directive transformations: perform basic transformations based on the code structure.
  - Loop transformation
  - Specific transformation
  - Accelerator abstraction/helper transformation
  - Utility transformation

1.1 Line continuation

CLAW directives can be defined on several lines using the continuation line symbol. The syntax is described as follows:

```
!$claw directive clause &
!$claw clause
```

1.2 Interpretation order of the CLAW directives

The CLAW transformations can be combined together. For example, `loop-fusion` and `loop-interchange` can be used together in a group of nested loops.

The default application order of the transformations is defined as follows:

1. `ignore`
2. `remove`
3. `primitive`

4. `array-transform`
5. `loop-extract`
6. `loop-hoist`
7. `loop-fusion`
8. `loop-interchange`
9. `on-the-fly`
10. `kcache`
11. `if extract`
12. `parallelize`
13. `parallelize forward`
14. formatting transformation (internal use transformation)

Users must be aware that transformations are applied sequentially and therefore, a transformation can be performed on code that has already been transformed.

1.3 Accelerator compile guard

As CLAW generates accelerator directives, it might be useful sometimes to protect the compilation of source code including accelerator directives as it might be wrong if the CLAW FORTRAN Compiler didn't process it.

For this reason, the CLAW language has a compile guard that is removed by the CLAW FORTRAN Compiler while it applies the other transformations. This compile guard is designed to make the compilation failed when transformations are not applied and accelerator directives are present in the source code.

The compile guard is shown in the listing below. It uses the target accelerator language with a specific claw directive. Example for OpenACC on line 2 and OpenMP on line 3.

```
1 !$<accelerator language prefix> claw-guard
2 !$acc claw-guard
3 !$omp claw-guard
```

This line is then removed once the transformations are applied.

2 CLAW high-level directives

2.1 Single Column Abstraction (SCA)

The `sca` directive is used to parallelize a single column based algorithm. In weather prediction models, the physical parametrizations are column independent problems. This means that the algorithm to compute the different output fields, can be defined with as a single do statement iterating over the vertical dimension and there is no horizontal dependency. The `sca` directive is then used to parallelize efficiently this column algorithm over the domain (horizontal dimensions). The directive is local to a subroutine. Therefore, as a pre-condition, the column algorithm must be enclosed in a module subroutine.

```
!$claw define dimension dim_id(lower_bound:upper_bound) &
[!$claw define dimension dim_id(lower_bound:upper_bound) &] ...
!$claw sca [data(var_1[,var_2] ...)] [over (dim_id|:[,dim_id|:] ...)]
```

Options and details

- *define dimension*: Define a new dimension on which the column model will be parallelized. The lower bound and upper bound of the defined dimension can be either an integer constant or an integer variable. If a variable is given, it will be added in-order to the signature of the subroutine as an `INTENT(IN)` parameter.
- *data (optional)*: Define a list of variables to be promoted and bypass the automatic promotion deduction.
- *over (optional)*: Define the location of the new dimensions in promoted variables. The : symbol reflects the position of the current dimensions before the promotion.

If the `data` clause is omitted, the automatic promotion of variables is done as follows:

- Array variables declared with the `INOUT` or `OUT` intents are automatically promoted with the defined dimensions. In function of the target, intermediate scalar or array variables may also be promoted.
- Scalar/array variables placed on the left hand-side of an assignment statement will be promoted if the right hand-side references any variables already promoted. If the `over` clause is omitted, the dimensions are added on the left-side following the the definition order if there is multiple new dimensions. The `over` clause allows to change this behavior and to place the new dimensions where needed.

As the subroutine/function signature is updated by the `sca` directive with new dimensions, the call graph that leads to this subroutine/function must be updated as well. For this purpose, the clause `forward` allows to replicated the changes along the call graph. Each subroutine/function along the call graph must be decorated with this directive.

```
! Call of a sca subroutine
!$claw sca forward
CALL one_column_fct(x,y,z)

! Call of a sca function
!$claw sca forward
result = one_column_fct(x,y,z)
```

The different variables declared in the current function/subroutine might be promoted if the call requires it. In this case, the array references in the function/subroutine are updated accordingly. This can lead to extra promotion of local scalar variables or arrays when needed.

The root function call in the call graph might be an iteration over several column to reproduce the algorithm on the grid for testing purpose. If the directive with the `forward` clause is placed just before one or several nested do statements with a function call, the corresponding do statements will be removed and the function call updated accordingly.

```
!$claw sca forward
DO i=istart, iend
  CALL one_columne_fct(x,y(i,:),z(i,:))
END DO
```

The root function call can also introduce blocking. For this, the `blocked` clause must be added to the directive. The `blocked` clause takes an argument that can be a positive integer or an `INTEGER` variable.

```
!$claw parallelize forward blocked(bsize)
DO p = 1, nproma
  CALL compute_column(nz, q(p,:), t(p,:), z(p,:))
END DO
```

The transformed code will automatically compute block sizes and split the horizontal dimension accordingly.

```
p_blocks = <number of blocks as a function of "bsize">
DO p_i = 1, p_blocks
   p0 = <first index of block in global array>
   p1 = <final index of block in global array>
   CALL compute_column ( nz , q (p0:p1 , : ) , t (p0:p1, : ) , z (p0:p1, : ), &
     nproma=p1-p0 )
END DO
```

Computation of the number of blocks is given by the implementation. User can also provide a function to perform this computation. This function must takes an `INTEGER` argument and return an `INTEGER` value.

To provide this function, the user can use the `block-fct(<function-name>)` clause.

Code example
Simple example of a column model wrapped into a subroutine and parallelized with CLAW.

Original code

```fortran
PROGRAM model
  USE mo_column, ONLY: compute_column

  REAL, DIMENSION(20,60) :: q, t  ! Fields as declared in the whole model
  INTEGER :: nproma, nz           ! Size of array fields
  INTEGER :: p                    ! Loop index

  nproma = 20
  nz = 60

  DO p = 1, nproma
    q(p,1) = 0.0
    t(p,1) = 0.0
  END DO

  ! Root call to the parallelized subroutine will be transformed as well.
    Arrays
  ! q and t will be passed entierly and the do statement will be removed.

  !$claw sca forward
  DO p = 1, nproma
    CALL compute_column(nz, q(p,:), t(p,:))
  END DO

  PRINT*,SUM(q)
  PRINT*,SUM(t)
END PROGRAM model
```

```fortran
MODULE mo_column
  IMPLICIT NONE
CONTAINS
  ! Compute only one column
  SUBROUTINE compute_column(nz, q, t)
    IMPLICIT NONE

    INTEGER, INTENT(IN)    :: nz   ! Size of the array field
    REAL, INTENT(INOUT)    :: t(:) ! Field declared as one column only
    REAL, INTENT(INOUT)    :: q(:) ! Field declared as one column only
    INTEGER :: k                   ! Loop index
    REAL :: c                      ! Coefficient
    REAL :: d                      ! Intermediate variable

    ! CLAW definition
    ! Define the new dimension on which the transformation parallelize the
    ! single column abstraction. In this case, the automatic promotion
    ! deduction will be activated as the data/over clauses are not present.

    !$claw define dimension proma(1:nproma) &
    !$claw sca

    c = 5.345
    DO k = 2, nz
      t(k) = c * k
      d    = t(k)**2
      q(k) = q(k - 1)  + t(k) * c
    END DO
    q(nz) = q(nz) * c
  END SUBROUTINE compute_column
END MODULE mo_column
```

Transformed code
Only the code of the module is shown as it is the only one which changes.

CLAW generated code for GPU target with OpenACC accelerator directive language.

```
MODULE mo_column

CONTAINS
 SUBROUTINE compute_column ( nz , q , t , nproma )
  INTEGER , INTENT(IN) :: nz
  REAL , INTENT(INOUT) :: t ( : , : )
  REAL , INTENT(INOUT) :: q ( : , : )
  INTEGER :: k
  REAL :: c
  REAL :: d

  INTEGER , INTENT(IN) :: nproma
  INTEGER :: proma


!$acc data present(q,nproma,nz,t)
!$acc parallel private(k,proma,d,c)
!$acc loop
  DO proma = 1 , nproma , 1
   c = 5.345
!$acc loop seq
   DO k = 2 , nz , 1
    t ( proma , k ) = c * k
    d = t ( proma , k ) ** ( 2 )
    q ( proma , k ) = q ( proma , k - 1 ) + t ( proma , k ) * c
   END DO
   q ( proma , nz ) = q ( proma , nz ) * c
  END DO
!$acc end parallel
```

```fortran
!$acc end data
 END SUBROUTINE compute_column

END MODULE mo_column
```

CLAW generated code for CPU target with OpenMP accelerator directive language.

```fortran
MODULE mo_column

CONTAINS
 SUBROUTINE compute_column ( nz , q , t , nproma )
  INTEGER , INTENT(IN) :: nz
  REAL , INTENT(INOUT) :: t ( : , : )
  REAL , INTENT(INOUT) :: q ( : , : )
  INTEGER :: k
  REAL :: c
  REAL :: d ( 1 : nproma )

  INTEGER , INTENT(IN) :: nproma
  INTEGER :: proma


!$omp parallel
  c = 5.345
  DO k = 2 , nz , 1
!$omp do
   DO proma = 1 , nproma , 1
    t ( proma , k ) = c * k
   END DO
!$omp end do
!$omp do
   DO proma = 1 , nproma , 1
    d ( proma ) = t ( proma , k ) ** ( 2 )
   END DO
!$omp end do
!$omp do
   DO proma = 1 , nproma , 1
    q ( proma , k ) = q ( proma , k - 1 ) + t ( proma , k ) * c
   END DO
!$omp end do
  END DO
!$omp do
  DO proma = 1 , nproma , 1
   q ( proma , nz ) = q ( proma , nz ) * c
  END DO
!$omp end do
!$omp end parallel
 END SUBROUTINE compute_column

END MODULE mo_column
```

Compilation with the CLAW FORTRAN Compiler

```
# For GPU target with OpenACC directive
clawfc --target=gpu --directive=openacc -o mo_column.gpu_acc.f90 mo_column.f90
clawfc --target=gpu --directove=openacc -o main.claw.f90 main.f90

# For CPU target without directive
clawfc -t=cpu -d=none -o mo_column.cpu_omp.f90 mo_column.f90
clawfc -t=cpu -d=none -o main.claw.f90 main.f90
```

2.1.1 SCA directive in ELEMENTAL function

Single Column Abstraction can be used within an ELEMENTAL function or subroutine. Depending on the desired target, the function/subroutine can be transformed to fit the programming paradigm. Once transformed, the function/subroutine is not garuanteed to be PURE or ELEMENTAL anymore.

2.1.2 Automatic Data Movement Managment

Some targets need to take care of data movement between an host and a device. This data movment can be handled by CLAW using the followings clauses on `!$claw sca forward` directive.

- `create`: the create clause will generate the appropriate data allocation/deallocation before and after the `sca forward` directive.
- `update[(in|out)]`: the update clause will generate the appropriate update before and/or after the `sca forward` directive. The update clause accepts `in` or `out` optional options.

This clauses have no effect when the transformation target does not need a data movement management.

2.1.3 SCA model configuration

Instead of defining dimension per subroutine or function, one can define them for the whole model. These definitions have to be done in a specific configuration file as detailled in this section. The promotion of variables are based on defined layout also specified in the same configuration file.

When using a model configuration file, the is simplified to the minimum required as shown below:

```
!$claw sca [layout(layout_id)]
```

In addition to the `sca` directive, a new directive is used to specify which variables are ”model data”. Model data are variables that will be promoted during the transformation. The `model-data` directive is a block directive and has to be placed in the declaration section of the subroutine/function. One or more `model-data` block can be declared in a subroutine/function.

```
!$claw model-data

! Variable declaration

!$claw end model-data
```

The specification of the model configuration file are listed below:

- The model configuration file follows the TOML file format.
- The configuration is spearated in three tables: model, dimensions, layouts.

Model table
The model table contains all global information about the model and its configuration.

Dimensions table
The dimensions table contains all dimension definitions. Each definition can have the information listed here:

- *id*: the identifier of the dimension definition. It is unique in the model configuration.
- *size*: the size of the dimension as a lower and upper bound. The lower bound can be omitted and the default value 1 will be assigned. Upper bound is mandatory.
- *iteration*: iteration information of the dimension defined with a lower and upper bound as well as a step. All information are optional. If omitted, lower and upper bound are copied from size information and the default value 1 is assigned to the step.

Layouts table
The layout table contains all the different layout that can be applied for a specific target or a specific subroutine/function.

- *name*: the identifier of the layout definition. It is unique in the model configuration. `default` is the only mandatory layout definiton.
- *position*: list of dimension definition that defined the layout. The semi-colon (:) define already existing definition.

Listing 1: Example of SCA model configuration file.

```
# ModelX SCA configuration file
# This file is an example to show the different posibility in the configuration
# file

[model] # Definition of global model information
  name = "ModelX"

[[dimensions]] # Definition of dimensions that can be used in layouts
  id = "horizontal"
  [dimensions.size]
    lower = 1              # if not specified, 1 by default
    upper = "nproma"       # mandatory information
  [dimension.iteration]
    lower = "pstart" # if not specified size.lower by default
    upper = "pend"   # if not specified size.upper by default
    step = 1         # if not specified, 1 by default

[[dimensions]] # Add a new dimension in the hashtable
  id = "vertical"
  [dimensions.size]
    upper = "klev"

[[layouts]] # Definition of layouts and default layout for specific target
  id = "default" # mandatory layout, used if no specific target layout
                 # specified
  position = [ "horizontal", ":" ]

[[layouts]] # Add a new layout in the hashtable
  id = "cpu" # Target specific layout for CPU
  position = [ ":" , "horizontal" ]

[[layouts]] # Add a new layout in the hashtable
  id = "gpu" # Target specific layout for GPU
  position = [ "horizontal", ":" ]

[[layouts]] # Add a new layout in the hashtable
  id = "radiation" # Specialized layout, can be mentioned in the sca
                   # clause
  position = [ ":" , "horizontal" ]
```

3 CLAW low-level directives

3.1 Global clauses

The following clauses can be applied to any kind of low-level transformations.

3.1.1 Target clause

The target clause specify for which transformed target the transformation is active. If no target is specified, the transformation is active for all targets. Current target are `cpu,gpu,none`.

```
!$claw directive [target(specific_target)]
```

The *specific_target* is one of the value provided by the compiler.

3.2 Loop transformation

3.2.1 Loop interchange/reordering

```
!$claw loop-interchange [(induction_var[, induction_var] ...)]
```

Loop reordering is a common transformation applied on do statements when adding parallelization. This transformation is mainly used to improve the data locality and avoid the need to transpose arrays.

The **loop-interchange** directive must be placed just before the first do statement composing the nested do statements group. With a group of two nested do statements, the induction variable and iteration range of the first statement is swapped with the information of the second do statement (see example 3.2.1).

If the list of induction variables is specified, the do statements are reordered according to the order defined in the list (see example A.3).

Options and details

1. *induction_var*: induction variable of the do statement. The list gives the new order of the do statement after the transformation. For group of 2 nested do statements, this information is not necessary, as the new order is deducted.

Behavior with other directives
When the do statements to be reordered are decorated with additional directives, those directives stay in place during the code transformation. In other words, they are not reordered together with the do statements iteration ranges (see example A.4).

Limitations
Currently, the **loop-interchange** directive is limited to 3 level of nested do statements. More nested levels can be declared but the transformation is limited to the first 3 levels from the directive declaration.

Code example
Example with 2 levels of loop.

Original code

```
!$claw loop-interchange
DO i=1, iend
  DO k=1, kend
    ! loop body here
  END DO
END DO
```

Transformed code

```
! CLAW transformation (loop-interchange i < -- > k)
DO k=1, kend
  DO i=1, iend
    ! loop body here
  END DO
END DO
```

More code examples in the appendix. Example with 3 nested do statements (see example A.3). Example with others directives (see example A.4).

3.2.2 Loop jamming/fusion

```
!$claw loop-fusion [group(group_id)] [collapse(n)]
```

Loop jamming or fusion is used to merge 2 or more do statements together. Sometimes, the work performed in a single do statement is too small to create significant impact on performance when it is parallelized. Merging some do statements together create bigger blocks (kernels) to be parallelized.

If the *group* clause is not specified, all the do statements decorated with the directive in the same structured block and at the same depth in the AST will be merged together as a single do statement (see code example in this section).

If the *group* clause is specified, the loops are merged in-order with do statements sharing the same group in the current structured block (see example A.5).

The *collapse* clause is used to specify how many tightly nested do statements are associated with the **loop-fusion** construct (see example A.7). The argument to the *collapse* clause is a constant positive integer. If the *collapse* clause is not specified, only the do statement that follows immediately the directive is associated with the **loop-fusion** construct.

Options and details

- *group_id*: An identifier that represents a group of do statement in a structured block (see example A.5).

- *n*: A constant positive integer.

Behavior with other directives
When the do statement to be merged are decorated with other directives, only the directives on the first do statement of the merge group are kept in the transformed code.

Limitations
All the do statement within a group must share the same iteration range. If the *collapse* clause is specified, the do statements must share the same iteration range at the corresponding depth (see example A.7).

Code example
Example without clauses.

Original code

```
DO k=1, iend
  !$claw loop-fusion
  DO i=1, iend
    ! loop #1 body here
  END DO

  !$claw loop-fusion
  DO i=1, iend
    ! loop #2 body here
  END DO
END DO
```

Transformed code

```
DO k=1, iend
  ! CLAW transformation (loop-fusion same block group)
  DO i=1, iend
    ! loop #1 body here
    ! loop #2 body here
  END DO
END DO
```

More code examples in the appendix. Example with the `group` clause (see example A.5). Example with others directives (see example A.6). Example with the `collapse` clause (see example A.7).

3.2.3 Loop extraction

```
!$claw loop-extract range(range_info) [[map(var[,var]...:mapping[,mapping]...)
    [map(var[,var]...:mapping[,mapping]...)] ...] [fusion [group(group_id)]] [
    parallel] [acc(directives)]
```

Loop extraction can be performed on a subroutine/function call. The do statement corresponding to the defined iteration range is extracted from the subroutine/function and is wrapped around the subroutine/function call. In the transformation, a copy of the subroutine/function is created with the corresponding transformation (demotion) for the parameters.

Options and details

1. *range*: Correspond to the induction variable of the do statement to be extracted. Notation `i`.
2. *map*: Define the mapping of variable that are demoted during the loop extraction. As seen in the example 3.2.3, the two parameters (1 dimensional array) are mapped to a scalar with the induction variable *i*.
   (a) The *var* can be defined as two parts variable (e.g. `a/a1`). The first part is the function call part and refers to the variable as it is defined in the function call. The second part is the function definition part and refers to the name of the variable to be mapped as it defined in the function declaration. If a *var* is defined as a single part variable, the same name is used for both the function call and function definition part.
   (b) The *mapping* can be defined as two parts variable (e.g. `i/i1`). The first part is the function call part and refers to the mapping variable as it is defined in the function call. The second part is the function definition part and refers to the name of the mapping variable as it defined in the function declaration. If a *mapping* is defined as a single part mapping variable, the same name is used for both the function call and function definition part.
3. *fusion*: Allow the extracted loop to be merged with other loops. Options are identical with the `loop-fusion` directive.
4. *parallel*: Wrap the extracted loop in a parallel region.
5. *acc*: Add the accelerator directives to the extracted loop.

If the directive **loop-extract** is used for more than one call to the same subroutine, the extraction can generate 1 to N dedicated subroutines.

Behavior with other directives
Accelerator directives are generated by the transformation. If the function call was decorated with accelerator directives prior to the transformation, those directives stay in place.

Limitations
The do statement to be extracted must be fully embraced in the subroutine/function. This means that no statements must be oustide of the do statement.

Code example
Example without additional clauses.

Original code

```fortran
PROGRAM main
  !$claw loop-extract range(i) map(value1,value2:i)
  CALL xyz(value1, value2)
END PROGRAM main

SUBROUTINE xyz(value1, value2)
  REAL, INTENT (IN) :: value2(x:y), value2(x:y)

  DO i = istart, iend
    ! some computation with value1(i) and value2(i) here
  END DO
END SUBROUTINE xyz
```

Transformed code

```fortran
PROGRAM main
  !CLAW extracted loop
  DO i = istart, iend
    CALL xyz_claw(value1(i), value2(i))
  END DO
END PROGRAM main

SUBROUTINE xyz(value1, value2)
  REAL, INTENT (IN) :: value2(x:y), value2(x:y)

  DO i = istart, iend
    ! some computation with value1(i) and value2(i) here
  END DO
END SUBROUTINE xyz

!CLAW extracted loop new subroutine
SUBROUTINE xyz_claw(value1, value2)
  REAL, INTENT (IN) :: value1, value2
  ! some computation with value1 and value2 here
END SUBROUTINE xyz_claw
```

More code examples in the appendix. Example with the `fusion` clause (see example A.8). Example with a more complicated mapping (see example A.9).

3.2.4 Loop hoisting

```
!$claw loop-hoist(induction_var[[, induction_var] ...]) [reshape(array_name(
    target_dimension[,kept_dimensions]))] [interchange [(induction_var[[,
    induction_var] ...])]] [fusion [group(group_id)] [collapse(n)]] [cleanup[(
    acc|omp)]]

  ! structured block of code
```

```

!$claw end loop-hoist
```

The **loop-hoist** directive allow single or nested loops in a defined structured block to be merged together and to hoist the beginning of those nested loop just after the directive declaration. Loops with different lower-bound indexes can also be merged with the addition of an `IF` statement. This feature works only when the lower-bound are integer constant.

Options and details

1. *induction_var*: List of induction variables of the do statements to be hoisted.
2. *interchange*: Allow the group of hoisted loops to be reordered after the current transformation has been performed. Options are identical with the **loop-interchange** directive.
3. *reshape*: Reshape arrays to scalar or to array with fewer dimensions. The original declaration is replaced with the corresponding demoted declaration in the current block (function/module).
4. *cleanup*: The cleanup clause will remove any OpenACC or OpenMP pragmas already present in the hoit block. If OpenACC or OpenMP is not specified, both are removed.
    (a) *array_name*: Name of the array type to be reshaped.
    (b) *target_dimension*: Number of dimension after reshaping. 0 indicates that the type is reshapped to a scalar.
    (c) *kept_dimension*: comma separated list of integer that indicates which dimensions are kept when the array is not totally demoted to scalar. Dimension index starts at 1.
5. *fusion*: Allow the extracted loop to be merged with other loops after the current transformation has been performed. Options are identical with the `loop-fusion` directive.

Limitations
If do statements in the hoist group need the addition of a `IF` statement, the lower-bound must be an integer constant.

Code example
Example without clauses.

Original code

```
!$acc parallel loop gang vector collapse(2)
DO jt=1,jtend
  !$claw loop-hoist(j,i) interchange
  IF ( .TRUE. ) CYCLE
    ! outside loop statement
  END IF
  DO j=1,jend
```

```
    DO i=1,iend
      ! first nested loop body
    END DO
  END DO
  DO j=2,jend
    DO i=1,iend
      ! second nested loop body
    END DO
  END DO
  !$claw end loop-hoist
END DO
!$acc end parallel
```

Transformed code

```
!$acc parallel loop gang vector collapse(2)
DO jt=1,jtend
  DO i=1,iend
    DO j=1,jend
      IF ( .TRUE. ) CYCLE
        ! outside loop statement
      END IF
      ! first nested loop body
      IF(j>1) THEN
        ! second nested loop body
      END IF
    END DO
  END DO
END DO
!$acc end parallel
```

3.2.5 Conditional extraction from loop

```
!$claw if-extract
```

Conditional extraction from loop allows to extract a perfectly nested if statement from a do statement.

Options and details
None.

Behavior with other directives
None.

Limitations
If the condition uses the induction variable, the transformation cannot be performed.

Code example
Example without clauses.

Original code

```
!$claw if-extract
DO i = 1, 10
  IF (test) THEN
    PRINT *, 'Then body:', i
  ELSE
    PRINT *, 'Else Body', i
  END IF
END DO
```

Transformed code

```
IF (test) THEN
  DO i = 1, 10
    PRINT *, 'Then body:', i
  END DO
ELSE
  DO i = 1, 10
    PRINT *, 'Else Body', i
  END DO
END IF
```

3.2.6 Loop fission/splitting

```
!$claw loop-fission
```

Loop fission is used to split a do statement into multiple loops over the same iteration range with each taking only a part of the original loop's body. The goal is to break down a large loop body into smaller ones to achieve better utilization of locality of reference or also reduce pressure on the register usage.

Options and details None.

Behavior with other directives
None.

Code example

Original code

```
DO k = 1, kend

```

```
  !  Part 1 of the loop body

  !$claw loop-fission

  !  Part 2 of the loop body

END DO
```

Transformed code

```
DO k = 1, kend

  !  Part 1 of the loop body

END DO

DO k = 1, kend

  !  Part 2 of the loop body

END DO
```

3.3 Specific transformation

3.3.1 K caching (column caching)

```
!$claw kcache data(var_1[,var_2] ...) [offset(offset_1[,offset_2] ...)]] [init]
     [private]
```

In memory-bound problem, it might be useful to cache array values used several times during the current do statement iteration or for the next one.

The **kcache** directive is applied in a limited do statement block. It will cache the corresponding assigned value and update the array index in the block according to the given plus/minus offsets. If the array referenced in the data clause is assigned in the block, the caching will take its place (see example 3.3.1). Otherwise, an assignment statement is created to start the caching (see example A.1).

If the offset values are omitted, there are inferred from the dimension of the variable to be cached and set to 0.

Options and details

- *data*: The data clause specifies which array will be impacted by the column caching transformation. List of array identifiers.

- *offset*: Integer value separated by a comma that represents the offset at each dimension.
- *init*: If the *init* clause is specified, the cache variable will be initialized with the corresponding array value at the given offset during the 1st loop iteration.
- *private*: it declares that a copy of each item on the list will be created for each parallel gang on the accelerator. The list is the one specified on the *data* clause.

Code example
Where caching takes assignment place and `init` clause.

Original code

```
DO j1 = k1start, k1end
  array_u6(j1,k3end) = pc1(j1,kend) * p2(j1)
ENDDO

DO j3 = k3start + 1, k3end
  DO j1 = k1start, k1end
    ! More computation here with others variables

    value1 = 1.0_dp / (1.0_dp - p2(j1) * (pc2(j1,j3) * array6(j1,j3-1)  &
                                   +  pc2(j1,j3) * array8(j1,j3-1) ))
    value2 = p2(j1) * (pc2(j1,j3) * array6(j1,j3-1)                      &
                       + pc2(j1,j3) * array8(j1,j3-1))
    value6 = p1(j1) * (pc2(j1,j3) * array6(j1,j3-1)                      &
                        +pc2(j1,j3) * array8(j1,j3-1))
        !$claw kcache data(array6) offset(0 -1) init
    array6(j1,j3) = p2(j1) * pc1(j1,j3)                                  &
                     + value2 * array8(j1,j3) + value6 * array8(j1,j3)
        !$claw kcache data(array6, array8) offset(0 -1) init
    array8(j1,j3) = p2(j1) * pc1(j1,j3)                                  &
                     + value6 * array2(j1,j3) + value7 * array4(j1,j3)
  END DO
END DO
```

Transformed code

```
DO j1 = k1start, k1end
  array_u6(j1,k3end) = pc1(j1,kend) * p2(j1)
ENDDO

DO j1 = k1start, k1end
  DO j3 = k3start + 1, k3end
    IF (j3 = k3start + 1) THEN
      array6_k_m1 = array6(j1,j3-1)
      array8_k_m1 = array8(j1,j3-1)
    END IF
    ! More computation here with others variables

    value1 = 1.0_dp / (1.0_dp - p2(j1) * (pc2(j1,j3) * array6_k_m1    &
                                 +  pc2(j1,j3) * array8_k_m1 ))
    value2 = p2(j1) * (pc2(j1,j3) * array6_k_m1                        &
                      + pc2(j1,j3) * array8_k_m1
    value6 = p1(j1) * (pc2(j1,j3) * array6_k_m1                        &
                       +pc2(j1,j3) * array8_k_m1
        array6_k_m1 = p2(j1) * pc1(j1,j3)                                  &
                    + value2 * array8(j1,j3) + value6 * array8(j1,j3)
    array6(j1,j3) = array6_k_m1
    array8_k_m1 = p2(j1) * pc1(j1,j3)                                  &
                    + value6 * array2(j1,j3) + value7 * array4(j1,j3)
    array8(j1,j3) = array8_k_m1
  END DO
END DO
```

More code examples in the appendix. Example with assignment created during the transformation (see example A.1). Example with `init` and `private` clause (see example A.2).

3.3.2 On-the-fly computation (array access to function call)

```
!$claw call array_name=function_call(arg_list)
```

Sometimes, replacing access to pre-computed arrays with computation on-the-fly can increase the performance. It can reduce the memory access for memory-bound kernels and exploit some unused resources to execute the computation.

This transformation is local to the current do statement. Array access from the directive will be replaced.

Options and details

- *array_name*: Array identifier. References to this array will be replaced by the function call in the current structured block.
- *function_call*: Name of the function provided by the user to compute the value. Return value must be of the same type as accessed value.
- *arg_list*: Comma separated list of arguments to be passed to the function call.

3.4 Accelerator abstraction/helper transformation

3.4.1 Conditional primitive directives

```
1 !$claw primitive_prefix primitive_directives
2
3 ! Example for OpenACC
4 !$claw acc routine seq
5
6 ! Example for OpenMP
7 !$claw omp target enter data
```

Primitive compiler directive like OpenACC or OpenMP can be enabled conditionally during the transformation. Examples on line 4 and 7 are conditional directives. They will be transformed into standard OpenACC respectively OpenMP directives only if the primitive directive language is specified for the transformation.

3.4.2 Vector notation expansion

```
!$claw expand [induction(name [[,] name]...)] [fusion [group(group_id)]] [parallel] [acc([clause [[,] clause]...])]

  ! vector notation assignment(s)

[!$claw end expand]
```

Computations using the vector notation are not suitable to be parallelized with language like OpenACC. The **expand** directive allows to transform those notation with the corresponding do statements which are more suitable for parallelization.

The goal of this directive is to pass from an vector notation assignment like this:

```
!$claw expand
A(1:n) = A(1+m:n+m) + B(1:n) * C(n+1:n+n)
```

To a do statement statement like this:

```
DO i=1,n
  A(i) = A(i+m) + B(i) * C(n+i)
END DO
```

If the directive is used as a block directive, the assignments are wrapped in a single do statement if their induction range match.

```
!$claw expand
A(1:n) = A(1+m:n+m) + B(1:n) * C(n+1:n+n)
B(1:n) = B(1:n) * 0.5
!$claw end expand
```

To

```
DO i=1,n
  A(i) = A(i+m) + B(i) * C(n+i)
  B(i) = B(i) * 0.5
END DO
```

Options and details

1. *induction*: Allow to name the induction variable created for the do statement.
2. *fusion*: Allow the create do statement to be merged with other loops. Options are identical with the **loop-fusion** directive
3. *parallel*: Wrap the extracted loop in a parallel region.
4. *acc*: Define accelerator clauses that will be applied to the generated loops.

Behavior with other directives
Directives declared before the **expand** directive will be kept in the generated code.

Code example
Example with vector notation following the pragma statement.

Original code

```
SUBROUTINE vector_add
  INTEGER :: i = 10
  INTEGER, DIMENSION(0:9) :: vec1

  !$claw expand
  vec1(0:i) = vec1(0:i) + 10;
END SUBROUTINE vector_add
```

Transformed code

```
SUBROUTINE vector_add
  INTEGER :: claw_i
  INTEGER :: i = 10
  INTEGER, DIMENSION(0:9) :: vec1

  ! CLAW transformation vector notation to do statement
  DO claw_i = 0, i
    vec1(claw_i) = vec1(claw_i) + 10;
  END DO
END SUBROUTINE vector_add
```

More code examples in the appendix. Example with the induction and acc clauses (see example **??**). Example with fusion clause (see example **??**). Example with 2-dimensional arrayx (see example **??**).

3.5 Utility transformation

3.5.1 Remove

```
!$claw remove

  ! code block

[!$claw end remove]
```

The **remove** directive allows the user to remove section of code during the transformation process. This code section is lost in the transformed code.

Options and details
If the directive is directly followed by a structured block (`IF` or `DO`), the end directive is not mandatory (see example 3.5.1). In any other cases, the end directive is mandatory.

Code example

Original code

```
DO k=1, kend
  DO i=1, iend
    ! loop #1 body here
  END DO

  !$claw remove
  IF (k > 1) THEN
    PRINT*, k
  END IF

  DO i=1, iend
    ! loop #2 body here
  END DO
END DO
```

Transformed code

```
DO k=1, kend
  DO i=1, iend
    ! loop #1 body here
  END DO

  DO i=1, iend
    ! loop #2 body here
  END DO
END DO
```

More code examples in the appendix. Example with block remove (see example A.13).

3.5.2 Ignore

```
!$claw ignore

  ! code block

!$claw end ignore
```

The `ignore` directive allows the user to ignore section of code for the transformation process. This code section is retrieved in the transformed code. Any CLAW directives in an ignored code section is also ignored. This could be useful to workaround any compilers bug and/or problems.

Code example
In the example below, the first remove block is ignored and the second is not.

Original code

```
PROGRAM testignore

  !$claw ignore
  !$claw remove
  PRINT*,'These lines'
  PRINT*,'are ignored'
  PRINT*,'by the CLAW compiler'
  PRINT*,'but kept in the final transformed code'
  PRINT*,'with the remove directives.'
  !$claw end remove
  !$claw end ignore

  !$claw remove
  PRINT*,'These lines'
  PRINT*,'are not ignored.'
  !$claw end remove

END PROGRAM testignore
```

Transformed code

```
PROGRAM testignore

  !$claw remove
  PRINT*,'These lines'
  PRINT*,'are ignored'
  PRINT*,'by the CLAW compiler'
  PRINT*,'but kept in the final transformed code'
  PRINT*,'with the remove directives.'
  !$claw end remove

END PROGRAM testignore
```

3.5.3 Verbatim

```
!$claw verbatim IF (i > 5) THEN

  ! code block

!$claw verbatim END IF
```

The `verbatim` directive allows the user to insert code after the transformation process. This code is not analyzed by the compiler.

Code example
In the example below, the first remove block is ignored and the second is not.

Original code

```
PROGRAM testverbatim

  !$claw verbatim IF (.FALSE.) THEN
  PRINT*,'These lines'
  PRINT*,'are not printed'
  PRINT*,'if the the CLAW compiler has processed'
  PRINT*,'the file.'
  !$claw verbatim END IF

END PROGRAM testverbatim
```

Transformed code

```
PROGRAM testverbatim

IF (.FALSE.) THEN
PRINT*,'These lines'
PRINT*,'are not printed'
PRINT*,'if the the CLAW compiler has processed'
PRINT*,'the file.'
END IF

END PROGRAM testverbatim
```

A Code examples

A.1 kcache 2

Assignment is created during the transformation.

Original code

```
DO j3 = ki3sc+1, ki3ec
  !$claw kcache data(array2)
  var1 = x * y - array2(j1, j3)
  var2 = z * array2(j1, j3)
END DO
```

Transformed code

```
DO j3 = ki3sc+1, ki3ec
  array2_k = array2(j1,j3)
  var1 = x * y - array2_k
  var2 = z * array2_k
END DO
```

A.2 kcache 3

Using `init` and `private` clause

Original code

```
!$acc parallel
DO j3 = ki3sc+1, ki3ec
  var3 = array1(j1,j3-1)

  !$claw kcache data(array1) offset(0 -1) init private
  array1(j1,ki3sc) = x * y * z
  var1 = x * y - array1(j1, j3-1)
  var2 = z * array1(j1, j3-1)
END DO
!$acc end parallel
```

Transformed code

```
!$acc parallel private(array1_k_m1)
DO j3 = ki3sc+1, ki3ec
  IF (j3 == ki3sc+1) THEN
    array1_k_m1 = array1(j1,j3-1)
  END IF
  var3 = array1_k_m1
  array1_k_m1 = x * y * z
  array1(j1,ki3sc) = array1_k_m1
  var1 = x * y - array1_k_m1
  var2 = z * array1_k_m1
END DO
!$acc end parallel
```

A.3 loop-interchange 2

Example with 3 levels of loop.

Original code

```
!$claw loop-interchange (k,i,j)
DO i=1, iend     ! loop at depth 0
  DO j=1, jend   ! loop at depth 1
    DO k=1, kend ! loop at depth 2
      ! loop body here
    END DO
  END DO
END DO
```

Transformed code

```
! CLAW transformation (loop-interchange (k,i,j))
DO k=1, kend       ! loop at depth 2
  DO i=1, iend     ! loop at depth 0
    DO j=1, jend   ! loop at depth 1
      ! loop body here
    END DO
  END DO
END DO
```

A.4 loop-interchange 3

Example with OpenACC directives.

Original code

```
!$acc parallel
!$acc loop gang
!$claw loop-interchange
DO i=1, iend
  !$acc loop vector
  DO k=1, kend
    ! loop body here
  END DO
END DO
!$acc end parallel
```

Transformed code

```
! CLAW transformation (loop-interchange i < -- > k)
!$acc parallel
!$acc loop gang
DO k=1, kend
  !$acc loop vector
  DO i=1, iend
    ! loop body here
  END DO
END DO
!$acc end parallel
```

A.5 loop-fusion 2

Example using the *group* clause.

Original code

```
DO k=1, iend
  !$claw loop-fusion group(g1)
  DO i=1, iend
    ! loop #1 body here
  END DO

  !$claw loop-fusion group(g1)
  DO i=1, iend
    ! loop #2 body here
  END DO

  !$claw loop-fusion group(g2)
  DO i=1, jend
    ! loop #3 body here
  END DO

  !$claw loop-fusion group(g2)
  DO i=1, jend
    ! loop #4 body here
  END DO
END DO
```

Transformed code

```
DO k=1, iend
  ! CLAW transformation (loop-fusion group g1)
  DO i=1, iend
    ! loop #1 body here
    ! loop #2 body here
  END DO

  ! CLAW tranformation (loop-fusion group g2)
  DO i=1, jend
    ! loop #3 body here
    ! loop #4 body here
  END DO
END DO
```

A.6 loop-fusion 3

Example showing behavior with other directives (OpenACC).

Original code

```
!$acc parallel
!$acc loop gang
DO k=1, iend
  !$acc loop seq
  !$claw loop-fusion
  DO i=1, iend
    ! loop #1 body here
  END DO

  !$acc loop vector
  !$claw loop-fusion
  DO i=1, iend
    ! loop #2 body here
  END DO
END DO
!$acc end parallel
```

Transformed code

```
!$acc parallel
!$acc loop gang
DO k=1, iend
  ! CLAW transformation (loop-fusion same block group)
  !$acc loop seq
  DO i=1, iend
    ! loop #1 body here
    ! loop #2 body here
  END DO
END DO
!$acc end parallel
```

A.7 loop-fusion 4

Example using the *collapse* clause.

Original code

```
DO k=1, iend
  !$claw loop-fusion collapse(2)
  DO i=0, iend
    DO j=0, jend
      ! nested loop #1 body here
    END FO
  END DO

  !$claw loop-fusion collapse(2)
  DO i=0, iend
    DO j=0, jend
      ! loop #2 body here
    END DO
  END DO
END DO
```

Transformed code

```
DO k=1, iend
  ! CLAW transformation (loop-fusion collapse(2))
  DO i=0, iend
    DO j=0, jend
      ! nested loop #1 body here
      ! nested loop #2 body here
    END DO
  END DO
END DO
```

A.8 loop-extract 2

Example with *fusion* clause.

Original code

```
PROGRAM main
  !$claw loop-extract(i=istart,iend) map(value1,value2:i) fusion group(g1)
  CALL xyz(value1, value2)

  !$claw loop-fusion group(g1)
  DO i = istart, iend
    ! some computation here
    print*,'Inside loop', i
  END DO
END PROGRAM main

SUBROUTINE xyz(value1, value2)
  REAL, INTENT (IN) :: value2(x:y), value2(x:y)

  DO i = istart, iend
    ! some computation with value1(i) and value2(i) here
  END DO
END SUBROUTINE xyz
```

Transformed code

```
PROGRAM main
  !CLAW extracted loop
  DO i = istart, iend
    CALL xyz_claw(value1(i), value2(i))
    ! some computation here
    print*,'Inside loop', i
  END DO
END PROGRAM main

SUBROUTINE xyz(value1, value2)
  REAL, INTENT (IN) :: value2(x:y), value2(x:y)

  DO i = istart, iend
    ! some computation with value1(i) and value2(i) here
  END DO
END SUBROUTINE xyz

!CLAW extracted loop new subroutine
SUBROUTINE xyz_claw(value1, value2)
  REAL, INTENT (IN) :: value1, value2
  ! some computation with value1 and value2 here
END SUBROUTINE xyz_claw
```

A.9 loop-extract 3

Example with more complicated mapping.

Original code

```
PROGRAM main
  !$claw loop-extract(i=istart,iend) map(value1,value2:i/j)
  CALL xyz(value1, value2)
END PROGRAM main

SUBROUTINE xyz(value1, value2, j)
  INTGER, INTENT(IN) :: j
  REAL  , INTENT(IN) :: value2(x:y), value2(x:y)

  DO i = istart, iend
    ! some computation with value1(j) and value2(j) here
  END DO
END SUBROUTINE xyz
```

Transformed code

```
PROGRAM main
  !CLAW extracted loop
  DO i = istart, iend
    CALL xyz_claw(value1(i), value2(i))
  END DO
END PROGRAM main

SUBROUTINE xyz(value1, value2, j)
  INTGER, INTENT(IN) :: j
  REAL  , INTENT(IN) :: value2(x:y), value2(x:y)

  DO i = istart, iend
    ! some computation with value1(j) and value2(j) here
  END DO
END SUBROUTINE xyz

!CLAW extracted loop new subroutine
SUBROUTINE xyz_claw(value1, value2, j)
  INTGER, INTENT(IN) :: j
  REAL, INTENT (IN) :: value1, value2
  ! some computation with value1 and value2 here
END SUBROUTINE xyz_claw
```

A.10 array-transform 2

Example with *induction* and *acc* clauses

Original code

```
SUBROUTINE vector_add
```

```
  INTEGER :: i = 10
  INTEGER, DIMENSION(0:9) :: vec1

  !$acc parallel
  !$claw array-transform induction(myinduc) acc(loop)
  vec1(0:i) = vec1(0:i) + 10;

  !$claw array-transform acc(loop)
  vec1(0:i) = vec1(0:i) + 1;
  !$acc end parallel
END SUBROUTINE vector_add
```

Transformed code

```
SUBROUTINE vector_add
  INTEGER :: claw_i
  INTEGER :: myinduc
  INTEGER :: i = 10
  INTEGER, DIMENSION(0:9) :: vec1

  !$acc parallel

  ! CLAW transformation array notation vec1(0:i) to do loop
  !$acc loop
  DO myinduc = 0, i
    vec1(myinduc) = vec1(myinduc) + 10;
  END DO

  ! CLAW transformation array notation vec1(0:i) to do loop
  !$acc loop
  DO claw_i = 0, i
    vec1(claw_i) = vec1(claw_i) + 1;
  END DO

  !$acc end parallel
END SUBROUTINE vector_add
```

A.11 array-transform 3

Example with fusion clause

Original code

```
SUBROUTINE vector_add
  INTEGER :: i = 10
  INTEGER, DIMENSION(0:9) :: vec1
  INTEGER, DIMENSION(0:9) :: vec2

  !$claw array-transform induction(claw_i) fusion
  vec1(0:i) = vec1(0:i) + 10;

  !$claw array-transform induction(claw_i) fusion
  vec2(0:i) = vec2(0:i) + 1;
END SUBROUTINE vector_add
```

Transformed code

```
SUBROUTINE vector_add
  INTEGER :: claw_i
  INTEGER :: i = 10
  INTEGER, DIMENSION(0:9) :: vec1
  INTEGER, DIMENSION(0:9) :: vec2

  ! CLAW transformation array notation vec1(0:i) to do loop
  ! CLAW transformation array notation vec2(0:i) to do loop
  ! CLAW transformation fusion
  DO claw_i=0, i
    vec1(claw_i) = vec1(claw_i) + 10;
    vec2(claw_i) = vec2(claw_i) + 1;
  END DO
END SUBROUTINE vector_add
```

A.12 array-transform 4

Example with 2-dimensional arrays

Original code

```
SUBROUTINE vector_add
INTEGER :: i = 10
INTEGER, DIMENSION(0:10,0:10) :: vec1
INTEGER, DIMENSION(0:10,0:10) :: vec2

vec1(0:i,0:i) = 0;
vec2(0:i,0:i) = 100;

!$claw array-transform
vec1(0:i,0:i) = vec2(0:i,0:i) + 10
END SUBROUTINE vector_add
```

Transformed code

```
SUBROUTINE vector_add
INTEGER :: i = 10
INTEGER, DIMENSION(0:10,0:10) :: vec1
```

```
INTEGER, DIMENSION(0:10,0:10) :: vec2

vec1(0:i,0:i) = 0;
vec2(0:i,0:i) = 100;

DO claw_i = 0, i, 1
  DO claw_j = 0, i, 1
    vec1(claw_i,claw_j) = vec2(claw_i, claw_j) + 10
  END DO
END DO
END SUBROUTINE vector_add
```

A.13 remove 2

Original code

```
DO k=1, kend
  DO i=1, iend
    ! loop #1 body here
  END DO

  !$claw remove
  PRINT*, k
  PRINT*, k+1
  !$claw end remove

  DO i=1, iend
    ! loop #2 body here
  END DO
END DO
```

Transformed code

```
DO k=1, kend
  DO i=1, iend
    ! loop #1 body here
  END DO

  DO i=1, iend
    ! loop #2 body here
  END DO
END DO
```