Directive Language Specification

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

Last updated: September 24, 2018

v1.1.0

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.

3

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:

```
1  !$claw directive clause &
2  !$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.

```
1  !$claw define dimension dim_id(lower_bound:upper_bound) &
2  [!$claw define dimension dim_id(lower_bound:upper_bound) &] ...
3  !$claw parallelize [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.

```
1  ! Call of a sca subroutine
2  !$claw parallelize forward
3  CALL one_column_fct(x,y,z)
4
5  ! Call of a sca function
6  !$claw parallelize forward
7  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.

```
1  !$claw parallelize forward
2  DO i=istart, iend
3    CALL one_columne_fct(x,y(i,:),z(i,:))
4  END DO
```

Code example

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

Original code

```
1  PROGRAM model
2    USE mo_column, ONLY: compute_column
3
4    REAL, DIMENSION(20,60) :: q, t  ! Fields as declared in the whole model
5    INTEGER :: nproma, nz           ! Size of array fields
6    INTEGER :: p                    ! Loop index
7
8    nproma = 20
9    nz = 60
10
11   DO p = 1, nproma
12     q(p,1) = 0.0
13     t(p,1) = 0.0
14   END DO
15
16   ! Root call to the parallelized subroutine will be transformed as well.
          Arrays
17   ! q and t will be passed entierly and the do statement will be removed.
18   !$claw parallelize forward
19   DO p = 1, nproma
20     CALL compute_column(nz, q(p,:), t(p,:))
21   END DO
22
23   PRINT*,SUM(q)
24   PRINT*,SUM(t)
25 END PROGRAM model
```

```
1  MODULE mo_column
2    IMPLICIT NONE
3  CONTAINS
4    ! Compute only one column
5    SUBROUTINE compute_column(nz, q, t)
6      IMPLICIT NONE
7
8      INTEGER, INTENT(IN)   :: nz   ! Size of the array field
```

7

```
 9      REAL, INTENT(INOUT)    :: t(:) ! Field declared as one column only
10      REAL, INTENT(INOUT)    :: q(:) ! Field declared as one column only
11      INTEGER :: k                   ! Loop index
12      REAL :: c                      ! Coefficient
13      REAL :: d                      ! Intermediate variable
14
15      ! CLAW definition
16      ! Define the new dimension on which the transformation parallelize the
17      ! single column abstraction. In this case, the automatic promotion
18      ! deduction will be activated as the data/over clauses are not present.
19
20      !$claw define dimension proma(1:nproma) &
21      !$claw parallelize
22
23      c = 5.345
24      DO k = 2, nz
25        t(k) = c * k
26        d    = t(k)**2
27        q(k) = q(k - 1)  + t(k) * c
28      END DO
29      q(nz) = q(nz) * c
30    END SUBROUTINE compute_column
31 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.

```
 1 MODULE mo_column
 2
 3 CONTAINS
 4  SUBROUTINE compute_column ( nz , q , t , nproma )
 5    INTEGER , INTENT(IN) :: nz
 6    REAL , INTENT(INOUT) :: t ( : , : )
 7    REAL , INTENT(INOUT) :: q ( : , : )
 8    INTEGER :: k
 9    REAL :: c
10    REAL :: d
11
12    INTEGER , INTENT(IN) :: nproma
13    INTEGER :: proma
14
15
16 !$acc data present(q,nproma,nz,t)
17 !$acc parallel private(k,proma,d,c)
18 !$acc loop
19    DO proma = 1 , nproma , 1
20      c = 5.345
```

```
21   !$acc loop seq
22      DO k = 2 , nz , 1
23       t ( proma , k ) = c * k
24       d = t ( proma , k ) ** ( 2 )
25       q ( proma , k ) = q ( proma , k - 1 ) + t ( proma , k ) * c
26      END DO
27      q ( proma , nz ) = q ( proma , nz ) * c
28     END DO
29   !$acc end parallel
30   !$acc end data
31    END SUBROUTINE compute_column
32
33   END MODULE mo_column
```

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

```
1    MODULE mo_column
2
3    CONTAINS
4     SUBROUTINE compute_column ( nz , q , t , nproma )
5      INTEGER , INTENT(IN) :: nz
6      REAL , INTENT(INOUT) :: t ( : , : )
7      REAL , INTENT(INOUT) :: q ( : , : )
8      INTEGER :: k
9      REAL :: c
10     REAL :: d ( 1 : nproma )
11
12     INTEGER , INTENT(IN) :: nproma
13     INTEGER :: proma
14
15
16   !$omp parallel
17     c = 5.345
18     DO k = 2 , nz , 1
19   !$omp do
20      DO proma = 1 , nproma , 1
21       t ( proma , k ) = c * k
22      END DO
23   !$omp end do
24   !$omp do
25      DO proma = 1 , nproma , 1
26       d ( proma ) = t ( proma , k ) ** ( 2 )
27      END DO
28   !$omp end do
29   !$omp do
30      DO proma = 1 , nproma , 1
31       q ( proma , k ) = q ( proma , k - 1 ) + t ( proma , k ) * c
32      END DO
33   !$omp end do
34     END DO
```

```
35  !$omp do
36    DO proma = 1 , nproma , 1
37      q ( proma , nz ) = q ( proma , nz ) * c
38    END DO
39  !$omp end do
40  !$omp end parallel
41   END SUBROUTINE compute_column
42
43  END MODULE mo_column
```

Compilation with the CLAW FORTRAN Compiler

```
1  # For GPU target with OpenACC directive
2  clawfc --target=gpu --directive=openacc -o mo_column.gpu_acc.f90 mo_column.f90
3  clawfc -o main.claw.f90 main.f90
4
5  # For CPU target with OpenMP directive
6  clawfc --target=gpu --directive=openmp -o mo_column.cpu_omp.f90 mo_column.f90
7  clawfc -o main.claw.f90 main.f90
```

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.

```
1  !$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

```
1  !$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

```
1  !$claw loop-interchange
2  DO i=1, iend
3    DO k=1, kend
4      ! loop body here
5    END DO
6  END DO
```

Transformed code

```
1  ! CLAW transformation (loop-interchange i < -- > k)
2  DO k=1, kend
3    DO i=1, iend
4      ! loop body here
5    END DO
6  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

```
1  !$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

```
1  DO k=1, iend
2    !$claw loop-fusion
```

```
 3    DO i=1, iend
 4       ! loop #1 body here
 5    END DO
 6
 7    !$claw loop-fusion
 8    DO i=1, iend
 9       ! loop #2 body here
10    END DO
11  END DO
```

Transformed code

```
1  DO k=1, iend
2    ! CLAW transformation (loop-fusion same block group)
3    DO i=1, iend
4       ! loop #1 body here
5       ! loop #2 body here
6    END DO
7  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

```
1  !$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.

13

(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

```
1   PROGRAM main
2     !$claw loop-extract range(i) map(value1,value2:i)
3     CALL xyz(value1, value2)
4   END PROGRAM main
5
6   SUBROUTINE xyz(value1, value2)
7     REAL, INTENT (IN) :: value2(x:y), value2(x:y)
8
9     DO i = istart, iend
10       ! some computation with value1(i) and value2(i) here
11     END DO
12   END SUBROUTINE xyz
```

Transformed code

```
1   PROGRAM main
2     !CLAW extracted loop
3     DO i = istart, iend
4       CALL xyz_claw(value1(i), value2(i))
```

```
5      END DO
6    END PROGRAM main
7
8    SUBROUTINE xyz(value1, value2)
9      REAL, INTENT (IN) :: value2(x:y), value2(x:y)
10
11     DO i = istart, iend
12       ! some computation with value1(i) and value2(i) here
13     END DO
14   END SUBROUTINE xyz
15
16   !CLAW extracted loop new subroutine
17   SUBROUTINE xyz_claw(value1, value2)
18     REAL, INTENT (IN) :: value1, value2
19     ! some computation with value1 and value2 here
20   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

```
1   !$claw loop-hoist(induction_var[[, induction_var] ...]) [reshape(array_name(
        target_dimension[,kept_dimensions]))] [interchange [(induction_var[[,
        induction_var] ...])]] [fusion [group(group_id)] [collapse(n)]] [cleanup[(
        acc|omp)]]
2
3     ! structured block of code
4
5   !$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

```
1   !$acc parallel loop gang vector collapse(2)
2   DO jt=1,jtend
3     !$claw loop-hoist(j,i) interchange
4     IF ( .TRUE. ) CYCLE
5       ! outside loop statement
6     END IF
7     DO j=1,jend
8       DO i=1,iend
9         ! first nested loop body
10      END DO
11    END DO
12    DO j=2,jend
13      DO i=1,iend
14        ! second nested loop body
15      END DO
16    END DO
17    !$claw end loop-hoist
18  END DO
19  !$acc end parallel
```

Transformed code

```
1   !$acc parallel loop gang vector collapse(2)
2   DO jt=1,jtend
3     DO i=1,iend
4       DO j=1,jend
5         IF ( .TRUE. ) CYCLE
```

```
 6            ! outside loop statement
 7         END IF
 8         ! first nested loop body
 9         IF(j>1) THEN
10            ! second nested loop body
11         END IF
12       END DO
13     END DO
14  END DO
15  !$acc end parallel
```

3.2.5 Conditional extraction from loop

```
 1  !$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

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

Transformed code

```
 1  IF (test) THEN
 2    DO i = 1, 10
 3      PRINT *, 'Then body:', i
 4    END DO
```

17

```
5   ELSE
6     DO i = 1, 10
7       PRINT *, 'Else Body', i
8     END DO
9   END IF
```

3.3   Specific transformation

3.3.1   K caching (column caching)

```
1   !$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

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

```
3   ENDDO
4
5   DO j3 = k3start + 1, k3end
6     DO j1 = k1start, k1end
7       ! More computation here with others variables
8
9       value1 = 1.0_dp / (1.0_dp - p2(j1) * (pc2(j1,j3) * array6(j1,j3-1)   &
10                                   + pc2(j1,j3) * array8(j1,j3-1) ))
11      value2 = p2(j1) * (pc2(j1,j3) * array6(j1,j3-1)                      &
12                       + pc2(j1,j3) * array8(j1,j3-1))
13      value6 = p1(j1) * (pc2(j1,j3) * array6(j1,j3-1)                      &
14                        +pc2(j1,j3) * array8(j1,j3-1))
15          !$claw kcache data(array6) offset(0 -1) init
16      array6(j1,j3) = p2(j1) * pc1(j1,j3)                                  &
17                      + value2 * array8(j1,j3) + value6 * array8(j1,j3)
18          !$claw kcache data(array6, array8) offset(0 -1) init
19      array8(j1,j3) = p2(j1) * pc1(j1,j3)                                  &
20                      + value6 * array2(j1,j3) + value7 * array4(j1,j3)
21    END DO
22  END DO
```

Transformed code

```
1   DO j1 = k1start, k1end
2     array_u6(j1,k3end) = pc1(j1,kend) * p2(j1)
3   ENDDO
4
5   DO j1 = k1start, k1end
6     DO j3 = k3start + 1, k3end
7       IF (j3 = k3start + 1) THEN
8         array6_k_m1 = array6(j1,j3-1)
9         array8_k_m1 = array8(j1,j3-1)
10      END IF
11      ! More computation here with others variables
12
13      value1 = 1.0_dp / (1.0_dp - p2(j1) * (pc2(j1,j3) * array6_k_m1    &
14                                  + pc2(j1,j3) * array8_k_m1 ))
15      value2 = p2(j1) * (pc2(j1,j3) * array6_k_m1                       &
16                       + pc2(j1,j3) * array8_k_m1
17      value6 = p1(j1) * (pc2(j1,j3) * array6_k_m1                       &
18                        +pc2(j1,j3) * array8_k_m1
19          array6_k_m1 = p2(j1) * pc1(j1,j3)                                &
20                      + value2 * array8(j1,j3) + value6 * array8(j1,j3)
21      array6(j1,j3) = array6_k_m1
22      array8_k_m1 = p2(j1) * pc1(j1,j3)                                 &
23                      + value6 * array2(j1,j3) + value7 * array4(j1,j3)
24      array8(j1,j3) = array8_k_m1
25    END DO
26  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)

```
1  !$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 Array notation to do statements

```
1 !$claw array-transform [induction(name [[,] name]...)] [fusion [group(group_id)
     ]] [parallel] [acc([clause [[,] clause]...])]
2
3   ! array notation assignment(s)
4
5 [!$claw end array-transform]
```

Computations using the array notation are not suitable to be parallelized with language like OpenACC. The **array-transform** 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 array notation assignment like this:

```
1 !$claw array-transform
2 A(1:n) = A(1+m:n+m) + B(1:n) * C(n+1:n+n)
```

To a do statement statement like this:

```
1 DO i=1,n
2   A(i) = A(i+m) + B(i) * C(n+i)
3 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.

```
1 !$claw array-transform
2 A(1:n) = A(1+m:n+m) + B(1:n) * C(n+1:n+n)
3 B(1:n) = B(1:n) * 0.5
4 !$claw end array-transform
```

To

```
1 DO i=1,n
2   A(i) = A(i+m) + B(i) * C(n+i)
3   B(i) = B(i) * 0.5
4 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.

21

Behavior with other directives

Directives declared before the **array-transform** directive will be kept in the generated code.

Code example

Example with array notation following the pragma statement.

Original code

```
1  SUBROUTINE vector_add
2    INTEGER :: i = 10
3    INTEGER, DIMENSION(0:9) :: vec1
4
5    !$claw array-transform
6    vec1(0:i) = vec1(0:i) + 10;
7  END SUBROUTINE vector_add
```

Transformed code

```
1   SUBROUTINE vector_add
2     INTEGER :: claw_i
3     INTEGER :: i = 10
4     INTEGER, DIMENSION(0:9) :: vec1
5
6     ! CLAW transformation array notation to do loop
7     DO claw_i = 0, i
8       vec1(claw_i) = vec1(claw_i) + 10;
9     END DO
10  END SUBROUTINE vector_add
```

More code examples in the appendix. Example with the induction and acc clauses (see example A.10). Example with fusion clause (see example A.11). Example with 2-dimensional arrayx (see example A.12).

3.5   Utility transformation

3.5.1   Remove

```
1  !$claw remove
2
3    ! code block
4
5  [!$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

```
1  DO k=1, kend
2    DO i=1, iend
3      ! loop #1 body here
4    END DO
5
6    !$claw remove
7    IF (k > 1) THEN
8      PRINT*, k
9    END IF
10
11   DO i=1, iend
12     ! loop #2 body here
13   END DO
14 END DO
```

Transformed code

```
1  DO k=1, kend
2    DO i=1, iend
3      ! loop #1 body here
4    END DO
5
6    DO i=1, iend
7      ! loop #2 body here
8    END DO
9  END DO
```

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

3.5.2   Ignore

```
1  !$claw ignore
2
3    ! code block
4
5  !$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

```
1   PROGRAM testignore
2
3     !$claw ignore
4     !$claw remove
5     PRINT*,'These lines'
6     PRINT*,'are ignored'
7     PRINT*,'by the CLAW compiler'
8     PRINT*,'but kept in the final transformed code'
9     PRINT*,'with the remove directives.'
10    !$claw end remove
11    !$claw end ignore
12
13    !$claw remove
14    PRINT*,'These lines'
15    PRINT*,'are not ignored.'
16    !$claw end remove
17
18  END PROGRAM testignore
```

Transformed code

```
1   PROGRAM testignore
2
3     !$claw remove
4     PRINT*,'These lines'
5     PRINT*,'are ignored'
6     PRINT*,'by the CLAW compiler'
7     PRINT*,'but kept in the final transformed code'
8     PRINT*,'with the remove directives.'
9     !$claw end remove
10
11  END PROGRAM testignore
```

3.5.3   Verbatim

```
1   !$claw verbatim IF (i > 5) THEN
2
3     ! code block
4
5   !$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

```
1  PROGRAM testverbatim
2
3    !$claw verbatim IF (.FALSE.) THEN
4    PRINT*,'These lines'
5    PRINT*,'are not printed'
6    PRINT*,'if the the CLAW compiler has processed'
7    PRINT*,'the file.'
8    !$claw verbatim END IF
9
10 END PROGRAM testverbatim
```

Transformed code

```
1  PROGRAM testverbatim
2
3  IF (.FALSE.) THEN
4  PRINT*,'These lines'
5  PRINT*,'are not printed'
6  PRINT*,'if the the CLAW compiler has processed'
7  PRINT*,'the file.'
8  END IF
9
10 END PROGRAM testverbatim
```

A   Code examples

A.1   kcache 2

Assignment is created during the transformation.

Original code

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

Transformed code

```
1  DO j3 = ki3sc+1, ki3ec
```

```
2    array2_k = array2(j1,j3)
3    var1 = x * y - array2_k
4    var2 = z * array2_k
5  END DO
```

A.2   kcache 3

Using `init` and `private` clause

Original code

```
1  !$acc parallel
2  DO j3 = ki3sc+1, ki3ec
3    var3 = array1(j1,j3-1)
4
5    !$claw kcache data(array1) offset(0 -1) init private
6    array1(j1,ki3sc) = x * y * z
7    var1 = x * y - array1(j1, j3-1)
8    var2 = z * array1(j1, j3-1)
9  END DO
10 !$acc end parallel
```

Transformed code

```
1  !$acc parallel private(array1_k_m1)
2  DO j3 = ki3sc+1, ki3ec
3    IF (j3 == ki3sc+1) THEN
4      array1_k_m1 = array1(j1,j3-1)
5    END IF
6    var3 = array1_k_m1
7    array1_k_m1 = x * y * z
8    array1(j1,ki3sc) = array1_k_m1
9    var1 = x * y - array1_k_m1
10   var2 = z * array1_k_m1
11 END DO
12 !$acc end parallel
```

A.3   loop-interchange 2

Example with 3 levels of loop.

Original code

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

```
3   DO j=1, jend    ! loop at depth 1
4     DO k=1, kend ! loop at depth 2
5       ! loop body here
6     END DO
7   END DO
8 END DO
```

Transformed code

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

A.4 loop-interchange 3

Example with OpenACC directives.

Original code

```
1  !$acc parallel
2  !$acc loop gang
3  !$claw loop-interchange
4  DO i=1, iend
5    !$acc loop vector
6    DO k=1, kend
7      ! loop body here
8    END DO
9  END DO
10 !$acc end parallel
```

Transformed code

```
1  ! CLAW transformation (loop-interchange i < -- > k)
2  !$acc parallel
3  !$acc loop gang
4  DO k=1, kend
5    !$acc loop vector
6    DO i=1, iend
7      ! loop body here
8    END DO
9  END DO
10 !$acc end parallel
```

A.5 loop-fusion 2

Example using the *group* clause.

Original code

```
1   DO k=1, iend
2     !$claw loop-fusion group(g1)
3     DO i=1, iend
4       ! loop #1 body here
5     END DO
6
7     !$claw loop-fusion group(g1)
8     DO i=1, iend
9       ! loop #2 body here
10    END DO
11
12    !$claw loop-fusion group(g2)
13    DO i=1, jend
14      ! loop #3 body here
15    END DO
16
17    !$claw loop-fusion group(g2)
18    DO i=1, jend
19      ! loop #4 body here
20    END DO
21  END DO
```

Transformed code

```
1   DO k=1, iend
2     ! CLAW transformation (loop-fusion group g1)
3     DO i=1, iend
4       ! loop #1 body here
5       ! loop #2 body here
6     END DO
7
8     ! CLAW tranformation (loop-fusion group g2)
9     DO i=1, jend
10      ! loop #3 body here
11      ! loop #4 body here
12    END DO
13  END DO
```

A.6 loop-fusion 3

Example showing behavior with other directives (OpenACC).

Original code

```
1  !$acc parallel
2  !$acc loop gang
3  DO k=1, iend
4    !$acc loop seq
5    !$claw loop-fusion
6    DO i=1, iend
7      ! loop #1 body here
8    END DO
9
10   !$acc loop vector
11   !$claw loop-fusion
12   DO i=1, iend
13     ! loop #2 body here
14   END DO
15 END DO
16 !$acc end parallel
```

Transformed code

```
1  !$acc parallel
2  !$acc loop gang
3  DO k=1, iend
4    ! CLAW transformation (loop-fusion same block group)
5    !$acc loop seq
6    DO i=1, iend
7      ! loop #1 body here
8      ! loop #2 body here
9    END DO
10 END DO
11 !$acc end parallel
```

A.7 loop-fusion 4

Example using the *collapse* clause.

Original code

```
1  DO k=1, iend
2    !$claw loop-fusion collapse(2)
3    DO i=0, iend
4      DO j=0, jend
5        ! nested loop #1 body here
6      END FO
7    END DO
8
9    !$claw loop-fusion collapse(2)
```

```
10    DO i=0, iend
11      DO j=0, jend
12        ! loop #2 body here
13      END DO
14    END DO
15  END DO
```

Transformed code

```
1  DO k=1, iend
2    ! CLAW transformation (loop-fusion collapse(2))
3    DO i=0, iend
4      DO j=0, jend
5        ! nested loop #1 body here
6        ! nested loop #2 body here
7      END DO
8    END DO
9  END DO
```

A.8   loop-extract 2

Example with *fusion* clause.

Original code

```
1  PROGRAM main
2    !$claw loop-extract(i=istart,iend) map(value1,value2:i) fusion group(g1)
3    CALL xyz(value1, value2)
4
5    !$claw loop-fusion group(g1)
6    DO i = istart, iend
7      ! some computation here
8      print*,'Inside loop', i
9    END DO
10 END PROGRAM main
11
12 SUBROUTINE xyz(value1, value2)
13   REAL, INTENT (IN) :: value2(x:y), value2(x:y)
14
15   DO i = istart, iend
16     ! some computation with value1(i) and value2(i) here
17   END DO
18 END SUBROUTINE xyz
```

Transformed code

```
1  PROGRAM main
```

```
 2    !CLAW extracted loop
 3    DO i = istart, iend
 4      CALL xyz_claw(value1(i), value2(i))
 5      ! some computation here
 6      print*,'Inside loop', i
 7    END DO
 8  END PROGRAM main
 9
10  SUBROUTINE xyz(value1, value2)
11    REAL, INTENT (IN) :: value2(x:y), value2(x:y)
12
13    DO i = istart, iend
14      ! some computation with value1(i) and value2(i) here
15    END DO
16  END SUBROUTINE xyz
17
18  !CLAW extracted loop new subroutine
19  SUBROUTINE xyz_claw(value1, value2)
20    REAL, INTENT (IN) :: value1, value2
21    ! some computation with value1 and value2 here
22  END SUBROUTINE xyz_claw
```

A.9   loop-extract 3

Example with more complicated mapping.

Original code

```
 1  PROGRAM main
 2    !$claw loop-extract(i=istart,iend) map(value1,value2:i/j)
 3    CALL xyz(value1, value2)
 4  END PROGRAM main
 5
 6  SUBROUTINE xyz(value1, value2, j)
 7    INTGER, INTENT(IN) :: j
 8    REAL  , INTENT(IN) :: value2(x:y), value2(x:y)
 9
10    DO i = istart, iend
11      ! some computation with value1(j) and value2(j) here
12    END DO
13  END SUBROUTINE xyz
```

Transformed code

```
 1  PROGRAM main
 2    !CLAW extracted loop
 3    DO i = istart, iend
 4      CALL xyz_claw(value1(i), value2(i))
```

```
 5    END DO
 6  END PROGRAM main
 7
 8  SUBROUTINE xyz(value1, value2, j)
 9    INTGER, INTENT(IN) :: j
10    REAL  , INTENT(IN) :: value2(x:y), value2(x:y)
11
12    DO i = istart, iend
13      ! some computation with value1(j) and value2(j) here
14    END DO
15  END SUBROUTINE xyz
16
17  !CLAW extracted loop new subroutine
18  SUBROUTINE xyz_claw(value1, value2, j)
19    INTGER, INTENT(IN) :: j
20    REAL, INTENT (IN) :: value1, value2
21    ! some computation with value1 and value2 here
22  END SUBROUTINE xyz_claw
```

A.10   array-transform 2

Example with *induction* and *acc* clauses

Original code

```
 1  SUBROUTINE vector_add
 2    INTEGER :: i = 10
 3    INTEGER, DIMENSION(0:9) :: vec1
 4
 5    !$acc parallel
 6    !$claw array-transform induction(myinduc) acc(loop)
 7    vec1(0:i) = vec1(0:i) + 10;
 8
 9    !$claw array-transform acc(loop)
10    vec1(0:i) = vec1(0:i) + 1;
11    !$acc end parallel
12  END SUBROUTINE vector_add
```

Transformed code

```
 1  SUBROUTINE vector_add
 2    INTEGER :: claw_i
 3    INTEGER :: myinduc
 4    INTEGER :: i = 10
 5    INTEGER, DIMENSION(0:9) :: vec1
 6
 7    !$acc parallel
 8
```

```
9    ! CLAW transformation array notation vec1(0:i) to do loop
10   !$acc loop
11   DO myinduc = 0, i
12     vec1(myinduc) = vec1(myinduc) + 10;
13   END DO
14
15   ! CLAW transformation array notation vec1(0:i) to do loop
16   !$acc loop
17   DO claw_i = 0, i
18     vec1(claw_i) = vec1(claw_i) + 1;
19   END DO
20
21   !$acc end parallel
22 END SUBROUTINE vector_add
```

A.11 array-transform 3

Example with fusion clause

Original code

```
1  SUBROUTINE vector_add
2    INTEGER :: i = 10
3    INTEGER, DIMENSION(0:9) :: vec1
4    INTEGER, DIMENSION(0:9) :: vec2
5
6    !$claw array-transform induction(claw_i) fusion
7    vec1(0:i) = vec1(0:i) + 10;
8
9    !$claw array-transform induction(claw_i) fusion
10   vec2(0:i) = vec2(0:i) + 1;
11 END SUBROUTINE vector_add
```

Transformed code

```
1  SUBROUTINE vector_add
2    INTEGER :: claw_i
3    INTEGER :: i = 10
4    INTEGER, DIMENSION(0:9) :: vec1
5    INTEGER, DIMENSION(0:9) :: vec2
6
7    ! CLAW transformation array notation vec1(0:i) to do loop
8    ! CLAW transformation array notation vec2(0:i) to do loop
9    ! CLAW transformation fusion
10   DO claw_i=0, i
11     vec1(claw_i) = vec1(claw_i) + 10;
12     vec2(claw_i) = vec2(claw_i) + 1;
13   END DO
```

```
14  END SUBROUTINE vector_add
```

A.12   array-transform 4

Example with 2-dimensional arrays

Original code

```
1   SUBROUTINE vector_add
2   INTEGER :: i = 10
3   INTEGER, DIMENSION(0:10,0:10) :: vec1
4   INTEGER, DIMENSION(0:10,0:10) :: vec2
5
6   vec1(0:i,0:i) = 0;
7   vec2(0:i,0:i) = 100;
8
9   !$claw array-transform
10  vec1(0:i,0:i) = vec2(0:i,0:i) + 10
11  END SUBROUTINE vector_add
```

Transformed code

```
1   SUBROUTINE vector_add
2   INTEGER :: i = 10
3   INTEGER, DIMENSION(0:10,0:10) :: vec1
4   INTEGER, DIMENSION(0:10,0:10) :: vec2
5
6   vec1(0:i,0:i) = 0;
7   vec2(0:i,0:i) = 100;
8
9   DO claw_i = 0, i, 1
10    DO claw_j = 0, i, 1
11      vec1(claw_i,claw_j) = vec2(claw_i, claw_j) + 10
12    END DO
13  END DO
14  END SUBROUTINE vector_add
```

A.13   remove 2

Original code

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

```
 5
 6    !$claw remove
 7    PRINT*, k
 8    PRINT*, k+1
 9    !$claw end remove
10
11    DO i=1, iend
12       ! loop #2 body here
13    END DO
14 END DO
```

Transformed code

```
1 DO k=1, kend
2    DO i=1, iend
3       ! loop #1 body here
4    END DO
5
6    DO i=1, iend
7       ! loop #2 body here
8    END DO
9 END DO
```