



# INCLUSIVE AND EXCLUSIVE SCAN

SSG DPD , Nikolay Panchenko

# Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright © 2016, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

## Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

# Agenda

- Scan
  - Motivating examples
  - Inclusive scan
    - Syntax
    - Semantics
    - Examples
  - Exclusive scan
    - Syntax
    - Semantics
    - Examples
  - User-Defined Scan
- Vectorizer features in 18.0
  - Explicit syntax for histogram-like pattern
  - Explicit syntax for compress and expand –like patterns
  - Explicit syntax for conditional lastprivates
  - Explicit vectorization of loops with breaks

# Motivating examples

## Inclusive scan

```
x = 0;
for (i = 0; i < n; ++i) {
    x += A[i];
    B[i] = x;
}
```

$B[j] = A[0] + A[1] + \dots + A[j-1] + A[j];$

## C++17:

```
template< class ExecutionPolicy, class ForwardIt1
, class ForwardIt2,
        class BinaryOperation, class T >
ForwardIt2
inclusive_scan(ExecutionPolicy&& policy, ForwardIt1 first, ForwardIt1 last, ForwardIt2
d_first, BinaryOperation binary_op, T init );
```

## Exclusive scan

```
x = 0;
for (i = 0; i < n; ++i) {
    B[i] = x;
    x += A[i];
}
```

$B[0] = 0;$

$B[j] = A[0] + A[1] + \dots + A[j-1], j > 0;$

## C++17:

```
template< class ExecutionPolicy, class ForwardIt1
, class ForwardIt2,
        class T, class BinaryOperation >
ForwardIt2
exclusive_scan(ExecutionPolicy&& policy, ForwardIt1 first, ForwardIt1 last, ForwardIt2
d_first, T init, BinaryOperation binary_op );
```

# Problems with automatic vectorization and/or parallelization

- In general, it's impossible to correctly identify synchronization point

```
for (i = 0; i < n; ++i) {  
    x += C[i];  
    foo(x, A[i]);  
    B[i] = x;  
}
```

---

- In general, it's impossible to transform exclusive scan to inclusive scan

```
for (i = 0; i < n; ++i) {  
    B[i] = x;  
    foo(x, A[i]);  
    x += C[i];  
}
```

# Inclusive scan definition

Definition of inclusive scan, of a variable  $X$  within the loop  $L$  : if two non-empty disjoint statement sequences  $S1$  and  $S2$  of  $L$  can be selected, such that

- $S1$  contains all definitions of  $X$ ; lexically last such statement is  $W$ ,
- $S1$  also contains all statements lexically prior to  $W$  that directly or indirectly use  $X$ ,
- Any definition in  $S1$   $X$  may only be used in  $S1$  except the definition of  $X$  in the statement  $W$ ,
- Direct or indirect use of  $X$  within  $S1$  must not appear in any condition expression,
- $S2$  consists of all statements lexically following  $W$ .

# Examples for the definition

## Good cases

## Bad cases

```
for (i = 0; i < n; ++i) {  
  x += A[i];  
  B[i] = x;  
}
```

```
for (i = 0; i < n; ++i) {  
  x += A[i];  
  if (i & 5) {  
    x = C[i];  
  }  
  B[i] = x;  
}
```

```
for (i = 0; i < n; ++i) {  
  t = x + A[i];  
  t1 = t;  
  x = t1;  
  if (x) {  
    B[i] = x;  
  }  
}
```

S1

S2



```
for (i = 0; i < n; ++i) {  
  x += A[i];  
  B[i] = x;  
  x += A[i];  
}
```

 backward dependency

```
for (i = 0; i < n; ++i) {  
  t = x + A[i];  
  C[i] = t;  
  x = t;  
  B[i] = x;  
}
```

 t lives only in S1  
 C[i] is live out

```
for (i = 0; i < n; ++i) {  
  x += A[i];  
  if (x > 9) {  
    t = C[i];  
  }  
  x += t;  
  B[i] = x;  
}
```

 depends on all updates of X  
 t lives only in S1

# Proposed syntax for inclusive scan

new clause: **scan**(*sc* : *item-list*)

- Loop level clause, which is used with `#pragma omp [simd | parallel for | ...]`
- *sc*: scan-combiner is any built-in binary operation (+, -, \*, ...) or UDS
- *item-list* contains one or more scalar variables

new pragma: `#pragma omp inclusive_scan(item-list)`

- “partition” pragma, which is used within the loop
- specifies a boundary between definitions and uses
- *item-list* contains one or more variables that are listed in the **scan** clause
- *item-list* cannot contain variables that are not listed in the **scan** clause
- a variable from *item-list* must not be used in any other construct
- the pragma must not be used in nested loops
- each list item from **scan** must be specified in only one **inclusive\_scan** (or **exclusive\_scan**)
- loop may contain multiple instances of **inclusive\_scan**

Aligned with `reduction/in_reduction` in OpenMP 5.0 Preview 2 (TR6) specification.



# Use of the syntax for simple case

```
x = 0;
#pragma omp simd scan(+: x)
for (i = 0; i < n; ++i) {
    x += A[i];
#pragma omp inclusive_scan(x)
    B[i] = x;
}
```

```
x = 0;
for (i = 0; i < n; i += 4) {
    vx = [0, 0, 0, 0];
    vx = vx + A[i + 3:i];
    rvx = [vx3 + vx2 + vx1 + vx0 + x,
          vx2 + vx1 + vx0 + x,
          vx1 + vx0 + x,
          vx0 + x];
    B[i + 3:i] = rvx;
    x = rvx3;
}
```

# Proposed semantics for inclusive scan

For the **scan** construct, a private copy  $vx$  of each list item is created, one for each SIMD lane as if the **private** clause had been used. On each iteration of the loop the private copy is initialized with identity value.

Any use of the list item prior to **inclusive\_scan** construct as if it is private.

A second private copy  $rvx$  for each list item is created at **inclusive\_scan** construct. On each iteration of the loop each SIMD lane of  $rvx$ ,  $rvx_i$ ,  $i \in \{0, \dots, vl - 1\}$ , is initialized as

$$rvx_i = sc(\dots (sc(sc(original_{item}, vx_0), vx_1) \dots vx_i), \forall i,$$

Any use of the list item after **inclusive\_scan** construct as if it is second private.

At the end of each iteration the original list item is updated by combining the original list item with the all lanes of the private copy using the combiner of the specified *scan-combiner*.

# Use of the syntax for simple case

For the scan construct, a private copy  $vx$  of each list item is created, one for each SIMD lane as if the **private** clause had been used. On each iteration of the loop the private copy is initialized with identity value.

Any use of the list item prior to **inclusive\_scan** construct as if it is private.

A second private copy  $rvx$  for each list item is created at **inclusive\_scan** construct. On each iteration of the loop each SIMD lane of  $rvx$ ,  $rvx_i$ ,  $i \in \{0, \dots, vl - 1\}$ , is initialized as

$$rvx_i = sc(\dots(sc(sc(original_{item}, vx_0), vx_1) \dots vx_i), \forall i,$$

Any use of the list item after **inclusive\_scan** construct as if it is second private.

At the end of each iteration the original list item is updated by  $rvx_{vl-1}$  which corresponds to the lexicographically last executed iteration of the loop.

```
x = 0;
#pragma omp simd scan(+: x)
for (i = 0; i < n; ++i) {
    x += A[i];
#pragma omp inclusive_scan(x)
    B[i] = x;
}
```

```
x = 0;
for (i = 0; i < n; i += 4) {
    vx = [0, 0, 0, 0];
    vx = vx + A[i + 3:i];
    rvx = [vx3 + vx2 + vx1 + vx0 + x,
          vx2 + vx1 + vx0 + x,
          vx1 + vx0 + x,
          vx0 + x];
    B[i + 3:i] = rvx;
    x = rvx3;
}
```

# Use of the syntax for simple case

For the **scan** construct, a private copy *vx* of each list item is created, one for each SIMD lane as if the **private** clause had been used. On each iteration of the loop the private copy is initialized with identity value.

Any use of the list item prior to **inclusive\_scan** construct as if it is private.

A second private copy *rvx* for each list item is created at **inclusive\_scan** construct. On each iteration of the loop each SIMD lane of *rvx*, *rvx<sub>i</sub>*,  $i \in \{0, \dots, vl - 1\}$ , is initialized as

$$rvx_i = sc(\dots(sc(sc(original_{item}, vx_0), vx_1) \dots vx_i), \forall i,$$

Any use of the list item after **inclusive\_scan** construct as if it is second private.

At the end of each iteration the original list item is updated by *rvx<sub>vl-1</sub>* which corresponds to the lexicographically last executed iteration of the loop.

```
x = 0;
#pragma omp simd scan(+: x)
for (i = 0; i < n; ++i) {
    x += A[i];
#pragma omp inclusive_scan(x)
    B[i] = x;
}
```

```
x = 0;
for (i = 0; i < n; i += 4) {
    vx = [0, 0, 0, 0];
    vx = vx + A[i + 3:i];
    rvx = [vx3 + vx2 + vx1 + vx0 + x,
          vx2 + vx1 + vx0 + x,
          vx1 + vx0 + x,
          vx0 + x];
    B[i + 3:i] = rvx;
    x = rvx3;
}
```

# Use of the syntax for simple case

For the **scan** construct, a private copy  $vx$  of each list item is created, one for each SIMD lane as if the **private** clause had been used. On each iteration of the loop the private copy is initialized with identity value. Any use of the list item prior to **inclusive\_scan** construct as if it is private.

A second private copy  $rvx$  for each list item is created at **inclusive\_scan** construct. On each iteration of the loop each SIMD lane of  $rvx$ ,  $rvx_i$ ,  $i \in \{0, \dots, vl - 1\}$ , is initialized as

$$rvx_i = sc(\dots(sc(sc(original_{item}, vx_0), vx_1) \dots vx_i), \forall i,$$

Any use of the list item after **inclusive\_scan** construct as if it is second private.

At the end of each iteration the original list item is updated by  $rvx_{vl-1}$  which corresponds to the lexicographically last executed iteration of the loop.

```
x = 0;
#pragma omp simd scan(+: x)
for (i = 0; i < n; ++i) {
    x += A[i];
#pragma omp inclusive_scan(x)
    B[i] = x;
}
```

```
x = 0;
for (i = 0; i < n; i += 4) {
    vx = [0, 0, 0, 0];
    vx = vx + A[i + 3:i];
    rvx = [vx3 + vx2 + vx1 + vx0 + x,
          vx2 + vx1 + vx0 + x,
          vx1 + vx0 + x,
          vx0 + x];
    B[i + 3:i] = rvx;
    x = rvx3;
}
```

# Use of the syntax for simple case

For the **scan** construct, a private copy  $vx$  of each list item is created, one for each SIMD lane as if the **private** clause had been used. On each iteration of the loop the private copy is initialized with identity value.

Any use of the list item prior to **inclusive\_scan** construct as if it is private.

A second private copy  $rvx$  for each list item is created at **inclusive\_scan** construct. On each iteration of the loop each SIMD lane of  $rvx$ ,  $rvx_i$ ,  $i \in \{0, \dots, vl - 1\}$ , is initialized as

$$rvx_i = sc(\dots(sc(sc(original_{item}, vx_0), vx_1) \dots vx_i), \forall i,$$

Any use of the list item after **inclusive\_scan** construct as if it is second private.

At the end of each iteration the original list item is updated by  $rvx_{vl-1}$  which corresponds to the lexicographically last executed iteration of the loop.

```
x = 0;
#pragma omp simd scan(+: x)
for (i = 0; i < n; ++i) {
    x += A[i];
#pragma omp inclusive_scan(x)
    B[i] = x;
}
```

```
x = 0;
for (i = 0; i < n; i += 4) {
    vx = [0, 0, 0, 0];
    vx = vx + A[i + 3:i];
    rvx = [vx3 + vx2 + vx1 + vx0 + x,
          vx2 + vx1 + vx0 + x,
          vx1 + vx0 + x,
          vx0 + x];
    B[i + 3:i] = rvx;
    x = rvx3;
}
```

# Use of the syntax for simple case

For the **scan** construct, a private copy  $vx$  of each list item is created, one for each SIMD lane as if the **private** clause had been used. On each iteration of the loop the private copy is initialized with identity value.

Any use of the list item prior to **inclusive\_scan** construct as if it is private.

A second private copy  $rvx$  for each list item is created at **inclusive\_scan** construct. On each iteration of the loop each SIMD lane of  $rvx$ ,  $rvx_i$ ,  $i \in \{0, \dots, vl - 1\}$ , is initialized as

$$rvx_i = sc(\dots(sc(sc(original_{item}, vx_0), vx_1) \dots vx_i), \forall i,$$

Any use of the list item after **inclusive\_scan** construct as if it is second private.

At the end of each iteration the original list item is updated by  $rvx_{vl-1}$  which corresponds to the lexicographically last executed iteration of the loop.

```
x = 0;
#pragma omp simd scan(+: x)
for (i = 0; i < n; ++i) {
    x += A[i];
#pragma omp inclusive_scan(x)
    B[i] = x;
}
```

```
x = 0;
for (i = 0; i < n; i += 4) {
    vx = [0, 0, 0, 0];
    vx = vx + A[i + 3:i];
    rvx = [vx3 + vx2 + vx1 + vx0 + x,
          vx2 + vx1 + vx0 + x,
          vx1 + vx0 + x,
          vx0 + x];
    B[i + 3:i] = rvx;
    x = rvx3;
}
```

# Programmer can cheat

```
x = 1;
#pragma omp simd scan(*: x)
for (i = 0; i < n; ++i) {
    x += i;
    x += A[i];
#pragma omp inclusive_scan(x)
    B[i] = x;
}
```

- This is legal
- Execution will be different to scalar execution
- Aligned with **reduction/in\_reduction (in TR6)** specification



# Use of the syntax with conditional updates

```
x = 0;
#pragma omp simd scan(+: x)
for (i = 0; i < n; ++i) {
    if (i & 5) {
        x += A[i];
    }
    else {
        x += 2 * A[i];
    }
}
#pragma omp inclusive_scan(x)
B[i] = x;
}
```

```
x = 0;
for (i = 0; i < n; i += 4) {
    vx = [0, 0, 0, 0];
    m = ([i + 3, i + 2, i + 1, i] & [5, 5, 5, 5])
    != 0;
    vx = vx + @m A[i + 3:i]@m;
    vx = vx + @m 2 * A[i + 3:i]@m;
    rvx = [vx3 + vx2 + vx1 + vx0 + x,
          vx2 + vx1 + vx0 + x,
          vx1 + vx0 + x,
          vx0 + x];
    B[i + 3:i] = rvx;
    x = rvx3;
}
```

# inclusive\_scan under condition

```
x = 0;
#pragma omp simd scan(+: x)
for (i = 0; i < n; ++i) {
    if (i & 5) {
        x += A[i];
    }
    #pragma omp inclusive_scan(x)
    B[i] = x;
}
```

```
x = 0;
for (i = 0; i < n; i += 4) {
    vx = [0, 0, 0, 0];
    m = ([i + 3, i + 2, i + 1, i] & [5, 5, 5, 5])
    != 0;
    vx = vx + @m A[i + 3:i]@m;
    // for all lanes of the rvx
    rvx = [vx3 + vx2 + vx1 + vx0 + x,
          vx2 + vx1 + vx0 + x,
          vx1 + vx0 + x,
          vx0 + x];
    B[i + 3:i] = @m rvx;
    x = rvx3;
}
```

# Use of the syntax for two scans

```
x = 0;
y = 0;
#pragma omp simd scan(+: x, y)
for (i = 0; i < n; ++i) {
    x += A[i];
#pragma omp inclusive_scan(x)
    B[i] = x;
    y += B[i];
#pragma omp inclusive_scan(y)
    C[i] = y;
}
```

```
x = 0;
y = 0;
for (i = 0; i < n; i += 4) {
    vx = [0, 0, 0, 0];
    vy = [0, 0, 0, 0];
    vx = vx + A[i + 3:i];
    rvx = [vx3 + vx2 + vx1 + vx0 + x,
           vx2 + vx1 + vx0 + x,
           vx1 + vx0 + x,
           vx0 + x];
    B[i + 3:i] = rvx;
    x = rxv3;
    vy = vy + B[i + 3:i];
    rvy = [vy3 + vy2 + vy1 + vy0 + y,
           vy2 + vy1 + vy0 + y,
           vy1 + vy0 + y,
           vy0 + y];
    C[i + 3:i] = rvy;
    y = rvy3;
}
```

# Non-conformant scan with outer loop

```
x = 0;
#pragma omp simd scan(+: x)
for (i = 0; i < n; ++i) {
    x += A[i];
    for (j = 0; j < m; ++j) {
        x += C[j];
    }
    #pragma omp inclusive_scan(x)
    B[j] = x;
}
```

In reality this loop is :

```
#pragma omp simd scan(+: x)
for (i = 0; i < n; ++i) {
    x += A[i];
    // j loop
    x += C[0];
    #pragma omp inclusive_scan(x)
    B[0] = x;
    x += C[1];
    #pragma omp inclusive_scan(x)
    B[1] = x;
    ...
    x += C[m - 1];
    #pragma omp inclusive_scan(x)
    B[m-1] = x;
}
```

which fall into non - conformant case.

So **#pragma omp inclusive\_scan** must be used in the same loop level as corresponding scan clause.

# Conformant scan with outer loop

```
x = 0;
#pragma omp simd scan(+: x)
for (i = 0; i < n; ++i) {
    x += A[i];
    for (j = 0; j < m; ++j) {
        x += C[j];
    }
#pragma omp inclusive_scan(x)
    B[j] = x;
}
```

```
x = 0;
for (i = 0; i < n; i += 4) {
    vx = [0, 0, 0, 0];
    vx = vx + A[i];
    for (j = 0; j < m; ++j) {
        vx += broadcast(C[j]);
    }
    rvx = [vx3 + vx2 + vx1 + vx0 + x,
          vx2 + vx1 + vx0 + x,
          vx1 + vx0 + x,
          vx0 + x];
    B[i + 3:i] = rvx;
    x = rvx3;
}
```

# Exclusive scan

## Simple exclusive scan

```
x = 0;
for (i = 0; i < n; ++i) {
    B[i] = x;
    x += A[i];
}

B[0] = 0;
B[j] = A[0]+A[1]+...+A[j-1], j > 0;
```

## Required transformation (exclusive -> inclusive scan)

```
x = 0;
for (i = 0; i < n; ++i) {
    x += A[i]; // With special
processing
    B[i] = x;
x += A[i];
}
```

Intel Confidential

# Exclusive scan definition

Definition of exclusive scan, of a variable  $X$  within the loop  $L$ : if two disjoint non-empty statement sequences  $S1$  and  $S2$  of  $L$  can be selected, such that

- Lexically last statement of  $S1$  precedes to lexically first statement of  $S2$
- $S2$  must contain at least one write to  $X$
- $S1$  must not have any writes to  $X$
- Any statement in  $L$  must belong to  $S1$  or  $S2$
- $S1$  and  $S2$  are independent, except dependencies to/from  $X$
- Direct or indirect use of  $X$  within  $S2$  must not appear in any condition expression

# Examples for the definition

## Good cases

## Bad cases

```
for (i = 0; i < n; ++i) {  
  B[i] = x;  
  x += A[i];  
}
```

```
for (i = 0; i < n; ++i) {  
  if (i & 5) {  
    C[i] = i;  
  }  
  B[i] = x;  
  D[i] = C[i];  
  x += A[i];  
}
```

```
for (i = 0; i < n; ++i) {  
  if (x) {  
    B[i] = x;  
  }  
  t = x + A[i];  
  t1 = t;  
  x = t1;  
}
```

S1

S2

```
for (i = 0; i < n; ++i) {  
  x += A[i];  
  B[i] = x;  
  x += A[i];  
}
```

```
for (i = 0; i < n; ++i) {  
  if (i & 5) {  
    C[i] = i;  
  }  
  B[i] = x;  
  x += A[i] + C[i];  
}
```

Forward dependency for C[]

```
for (i = 0; i < n; ++i) {  
  B[i] = x;  
  x += A[i];  
  if (x) {  
    t = C[i];  
  }  
  x += t;  
}
```

depends on all updates of X  
t lives only in S1



# Proposed syntax for exclusive scan

## **scan**(*sc* : *item-list*)

- Loop level clause, which is used with `#pragma omp [simd]`
- *sc*: scan-combiner is any built-in binary operation (+, -, \*, ...) or UDS
- *item-list* contains one or more scalar variables

## new directive: `#pragma omp exclusive_scan(item-list)`

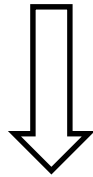
- “partition” pragma, which is used within the loop and allows to move block of statement before the pragma at the end of the loop or move block of statements after the pragma at the beginning of the loop
- *item-list* contains one or more variables that are listed in the **scan** clause
- *item-list* cannot contain variables that are not listed in the **scan** clause
- a variable from *item-list* must not be used in any other construct
- pragma must not be used under conditions or nested loops
- loop must have at most one instance of **exclusive\_scan**

**exclusive\_scan** pragma specifies end point of the block of statements that can be moved at the end of the loop.

**inclusive\_scan** pragma specifies place where running-reduction vector can be safely computed

# Use of the syntax for simple case

```
x = 0;
#pragma omp simd scan(+: x)
for (i = 0; i < n; ++i) {
    B[i] = x;
#pragma omp exclusive_scan(x)
    x += A[i];
}
```



Transformed by compiler

```
x = 0;
#pragma omp simd scan(+: x)
for (i = 0; i < n; ++i) {
    x += A[i];
#pragma omp
    __inclusive_scan_shifted_by_one(x)
    B[i] = x;
}
```

```
__inclusive_scan_shifted_by_one(x) =
[x2 + x1 + x0 + x_init,
 x1 + x0 + x_init,
 x0 + x_init,
 x_init];
```

```
inclusive_scan(x) =
[x3 + x2 + x1 + x0 + x_init,
 x2 + x1 + x0 + x_init,
 x1 + x0 + x_init,
 x0 + x_init];
```

# Proposed semantics for exclusive scan

For the **scan** construct, a private copy  $vx$  of each list item is created, one for each SIMD lane as if the **private** clause has been used. On each iteration of the loop the private copy is initialized with identity value.

Loop execution is changed as if each list item is transformed to **inclusive scan**.

A second private copy  $rvx$  for each list item is created at **exclusive\_scan** construct. On each iteration of the loop each SIMD lane of  $rvx$ ,  $rvx_i$ ,  $i \in \{0, \dots, vl - 1\}$ , is initialized as

$$rvx_0 = original_{item},$$
$$rvx_i = sc(\dots (sc(sc(original_{item}, vx_0), vx_1) \dots vx_{i-1}), \forall i \in \{1, \dots, vl - 1\},$$

Any use of the list item prior to **exclusive\_scan** construct as if it is second private.

At the end of each iteration the original list item is updated by combining the original list item with the all lanes of the private copy using the combiner of the specified *scan-combiner*.

# Find the right place for exclusive\_scan

```
x = 0;
#pragma omp simd scan(+:x)
for (i = 0; i < n; ++i) {
    if (A[i] > 0) {
        C[i] = x;
    }
#pragma omp exclusive_scan(x)
    D[i] = C[i];
#pragma omp exclusive_scan(x)
    if (A[i] < 0) {
        x += A[i];
    }
}
```

# Inclusive and Exclusive scans in the same loop

```
x = 0;
y = 0;
#pragma omp simd scan(+:x, y)
for (i = 0; i < n; ++i) {
    B[i] = x;
#pragma omp exclusive_scan(x)
    x += A[i];
    y += A[i];
#pragma omp inclusive_scan(y)
    C[i] = y;
}
```

# User-Defined Scan (UDS)

new pragma: `#pragma omp declare scan(scan-identifier : typename-list : combiner) [initializer-clause] new-line`

- Same as declare reduction except for **scan** keyword,
- *scan-identifier* is either a base language identifier or one of the following operations: +, -, \*, &, |, ^, && and ||
- *typename-list* is a list of type names
- *combiner* is an expression
- *initializer-clause* is **initializer(initializer-expr)** where *initializer-expr* is **omp\_priv = initializer** or **function-name(argument-list)**.

# Explicit syntax for histogram-like pattern

```
for (i = 0; i < n; ++i) {  
    ++a[b[i]];  
}
```

```
#pragma omp simd  
for (i = 0; i < n; ++i) {  
    #pragma omp ordered simd overlap(b[i])  
    {  
        ++a[b[i]];  
    }  
}
```

# Explicit syntax for compress-like pattern

```
for (i = 0; i < n; ++i) {  
    if (condition) {  
        a[++j] = b[i];  
    }  
}
```

```
#pragma omp simd  
for (i = 0; i < n; ++i) {  
    #pragma omp ordered simd monotonic(j:1)  
    {  
        if (condition) {  
            a[++j] = b[i];  
        }  
    }  
}
```



# Explicit syntax for expand-like pattern

```
for (i = 0; i < n; ++i) {  
    if (condition) {  
        b[i] = a[j++];  
    }  
}
```

```
#pragma omp simd  
for (i = 0; i < n; ++i) {  
    #pragma omp ordered simd monotonic(j:1)  
    {  
        if (condition) {  
            b[i] = a[j++];  
        }  
    }  
}
```

# Explicit syntax for conditional lastprivate

```
for (i = 0; i < n; ++i) {  
    if (condition) {  
        j = i;  
    }  
}
```

```
#pragma omp simd  
lastprivate(conditional: j)  
for (i = 0; i < n; ++i) {  
    if (condition) {  
        j = i;  
    }  
}
```

# Explicit syntax for loops with exits

```
for (i = 0; i < n; ++i) {  
    if (condition) {  
        j = i;  
        break;  
    }  
}
```

```
#pragma omp simd early_exit  
lastprivate(conditional: j)  
for (i = 0; i < n; ++i) {  
    if (condition) {  
        j = i;  
        break;  
    }  
}
```

