New Features in Offload Compiler
Agenda

• Variable references in input/output Clauses
• Data persistence and new pointer association model
• alloc & into modifiers
Variable References

- Objects named in `in / out / inout / nocopy` clauses were limited to whole variables
- New syntax allows struct member references and array slices
- Array slices must refer to a single contiguous block of memory
- `length` modifier can be replaced by array slice notation

### C / C++

```plaintext
variable-ref :
  identifier
  variable-ref . identifier
  variable-ref [ c-shape ]

c-shape :
  integral expression
  integral-expression [ : integral-expression ]
```

### Fortran

```plaintext
variable-ref :
  identifier
  variable-ref % identifier
  identifier ( ftn-shape )

ftn-shape :
  lower-bound-exp [ : upper-bound-exp ]
```
C/C++ Examples of Variable References

```c
01 typedef int ARRAY[10][10];
02 int a[1000][500];
03 int *p;
04 ARRAY *q;
05 int i, j;
06 struct { int y; } x;
10 #pragma offload … in( a )
11 #pragma offload … out( a[i:j][:] )
12 #pragma offload … in( p[0:100] )
13 #pragma offload … in( (*q)[5][:] )
14 #pragma offload … out( x.y )
```
Fortran Examples of Variable References

subroutine foo
real a(1000,500), b(1000,500), c(2000)
real, pointer :: p(:)
real, target :: t(1:20)
p => t(1:20)
!dec$ offload_transfer target(mic) in( a )
!dec$ offload_transfer target(mic) in( b(:,i:j) )
!dec$ offload_transfer target(mic) in( p(5:10) )
end
Data persistence

- Data is transferred from CPU to MIC in one offload and reused in a subsequent offload.

- Statically allocated variables are inherently persistent because MIC program stays loaded for the life of the CPU program.

- Cases of interest are pointers

```c
01 #define ALLOC alloc_if(1) free_if(0)
02 #define REUSE alloc_if(0) free_if(0)
03 #define FREE alloc_if(0) free_if(1)

10 int *p = malloc(...);
...
20 // Allocate memory on MIC, and transfer input data
21 #pragma offload_transfer ... in( p[0:100] ALLOC )
...
30 #pragma offload ... nocopy( p : REUSE )
31 { // use *p }
...
40 // Get results back and free memory on MIC
41 #pragma offload ... out( p[0:100] FREE )
```
Old pointer association model

- Persistent MIC memory is associated with a CPU *object*, not its value
- It is efficient, but has significant limitations
  - Once data is allocated, data transfer into MIC memory can only be done from that same object, referred by *name*
  - Assigning the CPU variable that tracks MIC memory to another variable does not lead to the assignee inheriting the MIC memory association

```c
int *p = malloc(...);
// Allocate memory on MIC, and transfer data
#pragma offload_transfer target(mic:0) in(p[0:100] : ALLOC)
foo(p, l);
```

---

```c
void foo(int *arg, ...) {
    // Transfer will fail because “arg” has no MIC-side allocation
    #pragma offload ... in(arg[0:100] : REUSE)
}
```
New pointer association model

13.0 Beta Update 2 implements new memory association model

Memory allocated on MIC from an “origin” CPU address is accessible through the same CPU address

Association is created / dropped along with MIC memory allocation / de-allocation

• Association is created when MIC memory is allocated
  – alloc_if(1) freeif(0)

• Association is deleted when MIC memory is freed
  – freeif(1)
Restrictions & Clarifications

One block of MIC memory associated with one CPU address

If an alloc_if(1) creates a second association for a CPU address before freeing the existing one, it is a user error
• The new association overwrites the earlier one
• This has the potential for causing memory leaks on MIC

If a free_if(1) is done for a CPU address and a matching table entry is not found, it is a user error
• The attempted removal is silently ignored
Example: Transfer through function parameter

Passing a pointer which has created an association to another function will work

```c
int *p = malloc(...);
int count;
void bar()
{
    ...
    // Allocate memory on MIC, and transfer data
    #pragma offload ... in( p[0:count] : ALLOC )
    foo(p, 1);
}

foo(int *arg_p, int count)
{
    // Transfer will succeed
    #pragma offload ... in( arg_p[0:count] : REUSE )
    ...
}
```
Changes to Existing Code

Existing code that depends on name association have to be changed

- While it is possible to support both “name” and “address” association we feel it is better to do one or the other for clarity

```c
int *p = malloc(...);  int *q = malloc(...);
int count;
// Allocate memory on MIC, and transfer data
#pragma offload ... in( p[0:count] : ALLOC )
...
// Old way to transfer different set of data will not work any more
p = q;
#pragma offload ... in( p[0:count] : REUSE )

// New way. p is associated with MIC preallocated memory
memcpy(p, q, ...);
#pragma offload ... in( p[0:count] : REUSE )
...
// Get results back and free memory on MIC
#pragma offload ... out( p[0:count] : FREE )
```
alloc Modifier

- alloc(<array-slice>) is an optional modifier used in in/out clause to specify a set of elements of the array that need allocation.

- It allows a smaller section of the array to be transferred to MIC without requiring that the entire array be allocated.

- Only unit stride is allowed in the <array-slice>

- When a section triplet has rank greater than one, the second and subsequent index expressions must specify all elements at that dimension.
Examples of Memory Allocation for Array Transfers

Example 1
int *p;
// 1000 elements allocated. Data transferred into p[10:100]
#pragma offload ... in ( p[10:100] : alloc(p[5:1000]) )
{ ... }

Example 2
typedef int ARRAY[4][4];
__declspec(target(mic)) ARRAY *p;

void f()
{
    p = (ARRAY*)malloc(16);

    // On MIC, 16 elements allocated
    // Array shape is 5x4, first row is unallocated.
    // Data is transferred into row 2 only
    #pragma offload_transfer target(mic) in ( (*p)[2][:] :

}
**into Modifier**

- **into**(variable-ref) is an optional modifier used in in/out clause that allows data to be transferred from one variable on the CPU to another on MIC, and vice versa.

- Only one item allowed in variable-list when using into modifier

- When used with in clause, copies from CPU object to MIC object
  - alloc_if / free_if / alloc modifiers apply to the into expression

- When used with out clause, copies from MIC object to CPU object
  - alloc_if / free_if / alloc modifiers apply to the out expression

- Disallowed with inout and nocopy clauses

- The source expressions generates a stream of elements to be copied into the memory range(s) specified by the into expression

- Overlap between source and destination leads to undefined behavior (although with disjoint memories it will work as expected)

- No ordering can be assumed between transfers from different in/out clauses

- The size of transferred data defined by in/out expression must be the same as in into one.
Examples of \texttt{into}

\begin{verbatim}
int p[1000], p1[2000];
int rank1[1000], rank2[10][100];

// Partial copy
#pragma offload ... in( p[0:500] : into p1[500:500] )

// Overlapping copy; result undefined
#pragma offload ... in( p[0:600] : into p1[0:600] )
\/

in( p[601:400] : into p1[100:400] )

// Shape change is not allowed
// Error!
#pragma offload ... out( rank1 : into(rank2) )
\end{verbatim}
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 © , Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Core, VTune, and Cilk
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