New MIC Offload Features in Alpha 9

Asynchronous Data Transfer

Asynchronous Compute

Memory management without data transfer

Available in both C/C++ and Fortran

- Examples are in C/C++ only

Two New Pragmas: Data Transfer and Wait for Completion

New pragma for data transfer only, with async option:

```
#pragma offload_transfer <clauses>
[ signal(<tag>) ]
```

New pragma to wait for completion of asynchronous activity

```
#pragma offload_wait <clauses> wait(<tag>)
```

Existing pragma takes optional signal/wait clauses

```
#pragma offload <clauses> [ signal(<tag>) ]
[ wait(<tag>) ]
    <statement>
```

New pragmas are stand-alone

Do not apply to the following statement, unlike `#pragma offload`

#pragma offload_transfer

A stand-alone pragma, i.e. no statement follows it

Contains `target` clause, and either all `in` clauses, or all `out` clauses

Without a `signal` clause, does synchronous data transfer

Can also take a `wait` clause

With a `signal` clause, *initiates* the data transfer only

- A later pragma with wait clause is used to wait for data transfer completion

Expressions in `signal/wait` clauses are address-sized values that serve as tags on the asynchronous operation

```
// Example 1:
// Synchronous data transfer CPU -> MIC
// Next statement executed after data transfer is completed
#pragma offload_transfer target(mic:0) in(a,b,c)

// Example 2:
// Initiate asynchronous data transfer CPU -> MIC
#pragma offload_transfer target(mic:0) in(a,b,c) signal(&a)
```

#pragma offload_wait

A stand-alone pragma, i.e. no statement follows it

Contains `target` clause

Contains a `wait` clause, which cause the pragma to start execution only after the async activity associated with the tag has completed

```
// Example 3:
// Wait for activity signaled by &p to be completed
#pragma offload_wait target(mic:0) wait(&p)
```

Allocating/Freeing MIC Memory

`#pragma offload_transfer` can be used for doing memory allocation *only* by avoiding the data transfer

Typically used outside loop to amortize cost of allocation

```
01 // Example 4:
02 #define ALLOC alloc_if(1) free_if(0)
03 #define FREE alloc_if(0) free_if(1)
04 #define REUSE alloc_if(0) free_if(0)
04 // Allocate memory on MIC (without also transferring data)
05 #pragma offload_transfer target(mic:0) nocopy(p,q : length(l) ALLOC)
06 …
10         for (…) {
11                   // Use of allocated memory on MIC for offloads
12                   #pragma offload target(mic:0) in(p:length(l) REUSE) out(q:length(l)
REUSE)
13                   { // computation using p and q }
14         }
15 …
20         // Free memory on MIC (without also transferring data)
21 #pragma offload_transfer target(mic:0) nocopy(p,q : length(l) FREE)
```

Send Input Data Asynchronously

Typical usage is to initiate data transfer, do some CPU activity, then start offload that will use the data

```
00 // Example 5:
01 // Initiate asynchronous data transfer MIC -> CPU
02 #pragma offload_transfer target(mic:0) in(p,q,r) signal(&p)
…
…
10 // Do the offload only after data has arrived
11 #pragma offload target(mic:0) wait(&p)
12 {
13                 // offload computation
14                 … = p;
15 }
```

The data is placed in the same variables listed in the transfer initiation

Those variables must be accessible at the second pragma

Receive Output Asynchronously

An offload produces results, but they will be transferred back later

The offload pragma finishes the compute but skips copying data back

An asynchronous pragma initiates the copy-back

Later, when results are needed, a `#pragma offload_wait` is used

```
00 // Example 6:
01 // Perform offload computation but don't copy back results immediately
02 #pragma offload target(mic:0) nocopy(p)
03 { p = …; }
04 // Initiate asynchronous data transfer MIC -> CPU
05 #pragma offload_transfer target(mic:0) out(p) signal(&p)
…
…
10 // Wait for data to arrive
11 #pragma offload_wait target(mic:0) wait(&p)
```

Asynchronous MIC Compute

```
00 // Example 6
01 char signal_var;
02 int *p;
03 do {
04        // Initiate asynchronous computation
05    #pragma offload … in( p:length(1000) ) signal(&signal_var)
06    {
07                mic_compute();
08    }
09    concurrent_cpu_activity();
10   #pragma offload_wait (&signal_var);
11 } while (1);
```

CPU initiates an offload to be done asynchronously

CPU can proceed to next statement after starting this computation

Later, an `offload_wait` pragma is used to wait for completion of the offload

API for Testing Signals

```
00 // Example 7:
01 // Initiate asynchronous computation
02 int c;
03 #pragma offload target(mic:mic_no) signal(&c) ...
04 { S3; }
...
10 // Test if computation has been completed
11 if _Offload_signaled(mic_no, &c) ….
```

Test whether the computation signaled with “c” is finished

Non-blocking mechanism to check if offload has been completed

Example 8: Doing an Offload Repeatedly

```
void do_sync()
{
        int i;
        for (i=0; i<iter; i++)
        {
                #pragma offload target(mic) \
                        in(in1   : length(count) REUSE ) \
                        out(out1 : length(count) REUSE )
                compute(in1, out1);
        }
}
```

Example 8: Double-buffering Input

```
void do_async_in()
{
  int i;
  #pragma offload_transfer target(mic:0) in(in1 : length(count) REUSE) signal(in1)
  for (i=0; i<iter; i++)
  {
    if (i%2 == 0) {
      #pragma offload_transfer target(mic:0) if(i!=iter-1) \
                      in(in2 : length(count) REUSE) signal(in2)
      #pragma offload target(mic:0) nocopy(in1) wait(in1)   \
                      out(out1 : length(count) REUSE)
      compute(in1, out1);
    } else {
      #pragma offload_transfer target(mic:0) if(i!=iter-1)
                      in(in1 : length(count) REUSE ) signal(in1)
      #pragma offload target(mic:0) nocopy(in2) wait(in2) \
                      out(out2 : length(count) REUSE)
      compute(in2, out2);
    }
  }
}
```

Example 8: Double-buffering Output

```
void do_async_out()
{
  int i;
  for (i=0; i<iter+1; i++) {
    if (i%2 == 0) {
      if (i<iter) {
        #pragma offload target(mic:0) in(in1 : length(count) REUSE) nocopy(out1)
          compute(in1, out1);
        #pragma offload_transfer target(mic:0) out(out1:length(count) REUSE) signal(out1)
      }
      if (i>0) {
        #pragma offload_wait target(mic:0) wait(out2)
        use_result(out2);
      }
    } else {
      if (i<iter) {
        #pragma offload target(mic:0) in(in2 : length(count) REUSE) nocopy(out2)
          compute(in2, out2);
        #pragma offload_transfer target(mic:0) out(out2:length(count) REUSE)) signal(out2)
      }
      if (i>0) {
        #pragma offload_wait target(mic:0) wait(out1)
        use_result(out1);
      }
    }
  }
}
```

Summary

Asynchronous offload allows overlap of data transfer and compute

Does not use additional CPU threads

Useful for pipelined operations

Available in Alpha 9