APPENDIX B

Virtual Shared Memory Programming on Xeon Phi

Although nonshared memory programming is widely used in developing applications for the Xeon Phi coprocessor, the Intel C/C++ compiler supports *virtual shared memory* programming. The advantage of this programming model is that it allows you to have more complex data structures shared between the host and client, removing the requirement that the data objects be bitwise copyable (such that the data can be copied using simple `memcpy` function) between the host and the coprocessor. With virtual shared memory constructs, the data copied between the host and coprocessor can be arbitrarily complex, including structures containing pointers such as linked list and tree data structures. The data must be placed in the shared virtual space before the offload computation can be performed on the shared virtual memory objects. In this model, the underlying software implements and maintains virtual memory that is shared between the host and the coprocessor.

Placing Data on the Virtual Shared Memory Region

In the virtual shared memory programming model, the underlying middle layer of the compiler runtime maintains memory regions that map to the same virtual address space for both the host and the coprocessor. The programming language extension by Intel C++ compiler defines a keyword `_Cilk_shared` that you can use to declare variables that the runtime should place in the virtual shared memory region. You can also define functions with the same keyword to indicate that these functions will be available on both the host and the coprocessor.

A variable declared with the `_Cilk_shared` keyword is placed in the same virtual memory address in both the host and the coprocessor, such that the pointer values are the same between the physically disjoint address spaces but give the appearance of being in the same address space. Thus, the offload code can work on non-bitwise copyable data structures such as linked list data structures. The runtime synchronizes the variables between these disjointed address spaces automatically at offload call sites.

When a variable is declared as shared, the memory is allocated dynamically for the variable by the host in the host and coprocessor virtual address spaces region that both share. As the declaration must be visible at the host compile time because the host manages the allocation of the shared variable, the declaration of such variables cannot be conditionally controlled. That is, you cannot declare such variables as follows:

```
#ifdef __ MIC__
_Cilk_shared float fvar1;
#endif
```

Incorrect code is generated here because during the host compile time the code generator does not see the variable declaration, as `__MIC__` macro is undefined. As such, the compiler will not allocate the shared memory address space needed by the variable in the coprocessor space. When the coprocessor tries to access the variable, an illegal access error will occur.

You can also use the `_Cilk_shared` keyword to declare the functions that will execute on the coprocessor. When a function is declared with a `_Cilk_shared` keyword, the code will generate two functions: one for the host and the other for the Xeon Phi coprocessor.

When applied to a C++ class, all of its member functions and all of the objects of that class are shared between the host and the coprocessor. You can apply the keyword to static fields of a class, assigning pointer-to-shared variables to pointer-to-nonshared variables. Other valid rules are assigning the address of a shared variable to a shared pointer. However, you cannot use `_Cilk_shared` on the field of a structure, on a static variable, or on variables local to a function.

For dynamic memory allocation in the virtual shared region, the compiler provides the following functions:

- `void *_Offload_shared_malloc(size_t size);` and `void *_Offload_shared_aligned_malloc(size_t size, size_t alignment);`: Allocates memory to be shared between the host and the coprocessor. This function reverts back to corresponding versions of `malloc()` if the coprocessor is not available or the Xeon Phi driver is not loaded.
- `void * _Offload_shared_free(void *p);` and `void * _Offload_shared_aligned_free(void *p);`: Frees the memory allocated by `Offload_shared_malloc` and `Offload_shared_aligned_malloc`. Reverts back to free if `malloc` is used during the allocation process, such as when the coprocessor is absent or the Xeon Phi driver was not loaded.

A third way of allocating virtual shared memory is to use the shared memory allocator defined in the Intel C++ standard library. By default, the containers provided by the standard C++ library will allocate memory from the nonshared memory space. When a standard C++ library object is declared with the `_Cilk_shared` keyword, its data members are allocated in the shared memory. However, these objects will use standard C++ allocators which will allocate memory in the nonshared memory space, resulting in a memory access error during the runtime as the memory space is adjusted for the object. The Intel C++ standard library provides a special allocator template for allocating memory from the shared address space declared as `shared_allocator<T>` defined in `offload.h`. For example, to be able to use a shared virtual memory space for a standard C++ library linked list class, you can define it as follows:

```
using namespace std;

typedef list <float, __offload::shared_allocator<float> > shared_list_float;

_Cilk_shared shared_list_float  * _Cilk_shared L;
```

Shared Functions

You can use the keyword `_Cilk_shared` to declare and define the functions that will be offloaded to Xeon Phi. The compiler creates two copies of the function: one for the Xeon host and the other for the Xeon Phi coprocessor. You can declare a shared function as `_Cilk_shared void f();` and define it as follows:

```
_Cilk_shared void f()
{
    x+y;
}
```

You can call these functions from host code to be offloaded to the coprocessor by using the `_Cilk_offload` keyword:

```
int main()
{
   _Cilk_offload f();
}
```

Some of the rules for using `_Cilk_offload` functions are as follows:

- The named function called by `_Cilk_offload f()` must be declared `_Cilk_shared` and defined `extern`.
- The function pointer in the `_Cilk_offload` indirect call must be of the type pointer-to-shared.
- A shared function whose address is taken must be defined `extern`.
- Pointer arguments sent to `offload_function` must be pointer-to-shared variables or objects.
- Global variables, pointers, and functions referenced within `_Cilk_offload _Cilk_for` must have shared attributes and pointers must point to shared variables.
- Global variables referenced inside the functions declared `_Cilk_shared` must be shared.

The `Cilk_offload` keyword allows synchronous and asynchronous execution of functions and code fragments on the CPU and coprocessor. Using `_Cilk_offload` before a function call executes the function on the coprocessor. Using the keyword `_Cilk_spawn` before `_Cilk_offload` results in asynchronous execution on the CPU and the coprocessor. You need to use the keyword `_Cilk_sync` before the host can use the result of the offloaded code for the asynchronous execution.

A `_Cilk_offload` before a `_Cilk_for loop` specifies that the entire loop is to be executed on the coprocessor. The syntax for using `_Cilk_offload` is as follows:

```
lvalue = _Cilk_offload func (rvalue)
lvalue = _Cilk_offload_to (target-number) func (rvalue)
lvalue = _Cilk_spawn _Cilk_offload func (rvalue)
lvalue = _Cilk_spawn _Cilk_offload_to (target-number) func (rvalue)
_Cilk_offload _Cilk_for (init-expr; test-expr; incr-expr;)
_Cilk_offload_to(target-number) _Cilk_for (init-expr; test-expr; incr-expr)
```

where:

`lvalue` is an object assigned the return value of the offload function call.
`rvalue` contains the input shared variables.
`func` is a function name to be offloaded.
`target-number` is a signed integer value having following interpretations:

> -1: The runtime system selects the target. If no coprocessor is available, the program fails with an error message.
>
> >=0: The code will be executed on the (coprocessor number = target-number %(modulo) total number of coprocessor). For example, if you have 2 coprocessors, the number 4 will signify mic0 (4 modulo 2) or the first coprocessor enumerated by the runtime system and so on.

`init-expr`, `test-expr`, and `incr-expr` are initialization, test, and increment expressions, respectively, for the `for` loop expression that will be offloaded to the coprocessor for execution.

Additional examples of virtual shared memory usage can be found under `compiler-install-dir/Samples/en_US/C++/mic_samples/LEO_tutorial`.

Synchronizing Between the Host and the Coprocessors

Memory synchronization for shared memory variables between the host and the coprocessor happens at the following points only:

1. When an offload function is called by the host code and is entering the offload function on the coprocessor
2. When an offload function on the target returns
3. When `_Cilk_sync` is invoked after an asynchronous offload using `_Cilk_spawn`

Any simultaneous access between the host and the coprocessor outside these synchronization points will result in a race condition, and the value of the shared variables will be undefined.

It is possible for multiple host threads to execute concurrently on the host, and for one or more host threads to offload computation to the coprocessor. In this case, you cannot synchronize between the offloaded code and other threads running on the host through thread synchronization mechanisms, such as mutexes and atomics. However, if the host uses OpenMP parallelism and one or more of the OpenMP threads perform offload, the OpenMP synchronization at the end of the OpenMP parallel region is guaranteed to work. An exception cannot be generated in the coprocessor to be caught and handled on the host.