Quick Start Guide for Offload Mode Execution

Arturo Argueta, Roberto Camacho Barranco, Esthela Gallardo and Pat Teller
UTEP Stampede Technology Insertion Project
Leonardo Fialho and Jim Browne
UT-Austin Stampede Technology Insertion Project

Xeon Phis can execute highly-parallel code segments (functions) of a program, while the remainder of the program is executed on the CPUs of a node, e.g., the 2 Sandy Bridge (SB) processors of a Stampede node. This is very similar to the approach used when accelerating program execution with GPGPUs.

The offloading of functions to Xeon Phis can be achieved by adding the following `pragma` statement to your program just before each section of code that you want to "offload" to a Phi:

```
#pragma offload target(mic)
```

Since OpenMP is used to execute offloaded code on Phi cores, when it is time to compile the code, an additional compiler flag, `-openmp`, is required. For example, to compile the program Example.c execute:

```
$ icc -openmp -o Example Example.c
```

When the program is executed, if a Phi is present on the node on which the program is executing, the annotated functions will be executed on it, otherwise they will be executed on the CPUs.

Below a sample program with a section of code that is to be offloaded to a Phi.

Using OpenMP in CPUs and Xeon Phis

If the program has an OpenMP region that will execute on the CPUs and another that will be offloaded to a Phi, two different OpenMP runtime environments will be created. Because of this, environment variables such as KMP_AFFINITY and OMP_NUM_THREADS will only be valid for the CPUs unless you change the default values for the Phis as follows.

To set `KMP_AFFINITY` on the CPUs use the following commands:

```
$ export KMP_AFFINITY=balanced
$ export OMP_NUM_THREADS=16
```

To set `KMP_AFFINITY` on Xeon Phis use the following commands:

```
$ export MIC_ENV_PREFIX=PHI
$ export PHI_KMP_AFFINITY=compact
$ export PHI_KMP_PLACE_THREADS=60c,3t
$ export PHI_OMP_NUM_THREADS=180
```

For more information regarding KMP_AFFINITY and thread affinity control see: *https://software.intel.com/en-us/articles/openmp-thread-affinity-control*.

Asynchronous Offload and Data Transfer

Using the default offload pragma, the host CPU will wait until the offload commands have been executed and the results are received. However, to achieve good performance, it may be necessary to employ asynchronous execution, i.e., have the CPU offload the computation and proceed immediately to the instruction following the offloaded region of code. Asynchronous offload will hide some of the cost of data transfers between the host and the Phi. It is provided by a special version of the offload pragma: a `signal(signal_value)` clause must be specified in the offload pragma and then a blocking `offload_wait` pragma or a non-blocking API `_offload_signaled()` can be used to wait until the offload is completed. For more information regarding asynchronous offload see: *https://software.intel.com/en-us/node/512566*. Note that even if asynchronous offload is employed, transferred data may need to be set up so that the Phi can utilize it, which has a cost in terms of execution time.

Sample Code

The following vector addition code is designed for execution in offload mode on the cores of a Phi. Execution begins on the host CPU, i.e., it creates the vectors a, b, and c. Then, after setting up structures for timing and initializing the vectors, the computational loop is executed on a Phi. To accomplish this, data is transferred between the SB CPU and the Phi (see *Offload Mode: Estimation of Execution Time and Cost of Data Movement* [URL]). Note the `offload` pragmas that demarcate the OpenMP parallel region that is meant to be offloaded to a Phi. The statements outside the demarcated sections of code are executed on the host CPU.

```
// Add two vectors in parallel using OpenMP.
#include <stdio.h>
#include <omp.h>
#include <time.h>
#include "offload.h"
#include <sys/time.h>

/* Global declaration of variables; modify “size” to study how
problem size, along with number of cores and threads per core,
affect execution time. */
__attribute__ ((target(mic))) const int size = 10000;
__attribute__ ((target(mic))) float a[size];
__attribute__ ((target(mic))) float b[size];
__attribute__ ((target(mic))) float c[size];
```

```c
int main()
{
    /* Timeval structures store the start and end time of
       initialization and computation. */
    struct timeval start, end;
    // Start timing vector initialization loop.
    gettimeofday(&start, NULL);

    // Initialize vectors.
    for (int i = 0; i < size; ++i) {
        a[i] = (float)i;
        b[i] = (float)i;
        c[i] = 0.0f;
    }

   // End timing vector initialization loop.
   gettimeofday(&end, NULL);
   // Print execution time of vector initialization loop.
   printf ("Total time before pragma = %f seconds\n",
           (double) (end.tv_usec - start.tv_usec) / 1000000 +
           (double) (end.tv_sec - start.tv_sec));

   // Start timing offloaded computational loop.
   gettimeofday(&start, NULL);

    /* Offload the following code to mic 0. Arrays a, b, c are
       transferred to the Phi and back to the SB. */
    // Compute vector add; C <- C = A + B
    #pragma offload  target(mic:0) inout(a,b,c)
    // Vector addition offloaded to mic 0.
    #pragma omp parallel for default(none) shared(a,b,c)
    for (int i = 0; i < size; ++i) {
          c[i] += a[i] + b[i];
    }

    // End timing offloaded computational loop.
    gettimeofday(&end, NULL);
    // Print execution time of offloaded computational loop.
    printf ("Total time after pragma = %f seconds\n",
           (double) (end.tv_usec - start.tv_usec) / 1000000 +
           (double) (end.tv_sec - start.tv_sec));

    return 0;
}
```