Agenda

- Intel MIC Architecture
- Models for Xeon Phi programming
  - Native
  - Offload

MIC: Why and what?

Key aspects of acceleration:

- An increasing number of transistors are being built into a processing chip; Moore's law is holding
- Clock rates have stalled at $<$4 GHz. Higher rates cause excess power consumption. (Denard's scaling no longer holds.)
- Only way to increase FLOPS per watt is through greater on-die parallelism
- This gives birth to the Many Integrated Core (MIC) architecure. In the high performance computing (HPC) market, Intel MIC is a direct competitor to the NVIDIA GPUs.

Intel Xeon Phi Coprocessors and MIC Architecture

- PCIe end-point device
- High Power efficiency
- 1 TFLOP/s in DP
- Heterogeneous clustering

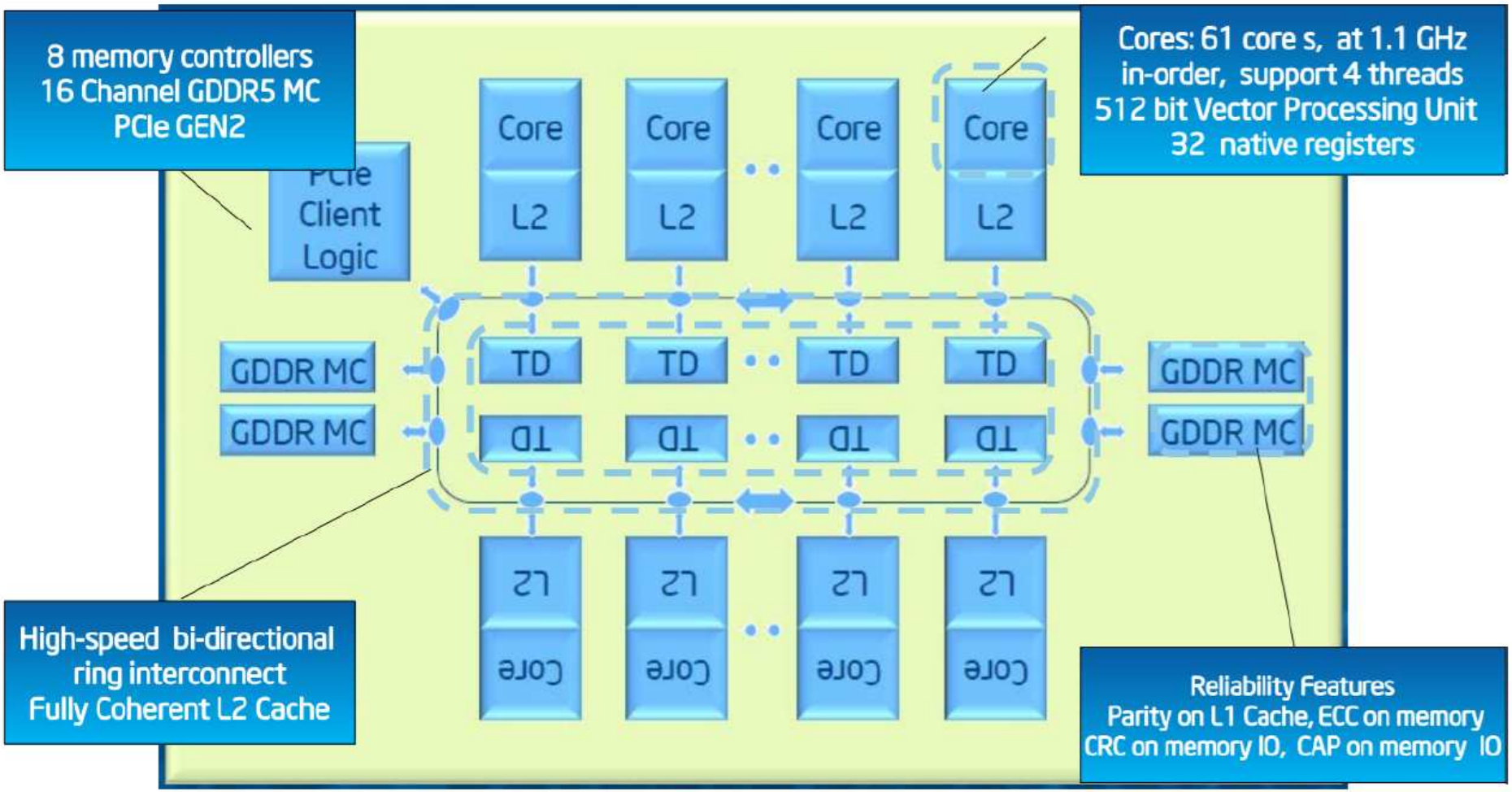
Intel MIC

- Xeon Phi = first (currently only) product of Intel's Many Integrated Core (MIC) architecture
- Usually run as a co-processor
  - PCI Express card
  - Stripped down Linux operating system, separate from host's
- Dense, simplified processor
  - Many power-hungry operations removed
  - Wider vector unit
  - Wider hardware thread count

Xeon Phi — MIC

- Based on x86 arcitecture (CPU with many cores)
  - x86 cores that are simpler, but allow for more compute throughput
- Can reuse existing x86 programming models
- Dedicate much of the silicon to floating point operations, increasing floating-point throughput
- Cache coherent
- Strip expensive features
  - No out-of-order execution (compiler interleaves computations)
  - No branch prediction (compiler unrolls loops)
- Widen SIMD registers for more throughput
- Fast (GDDR5) memory on card

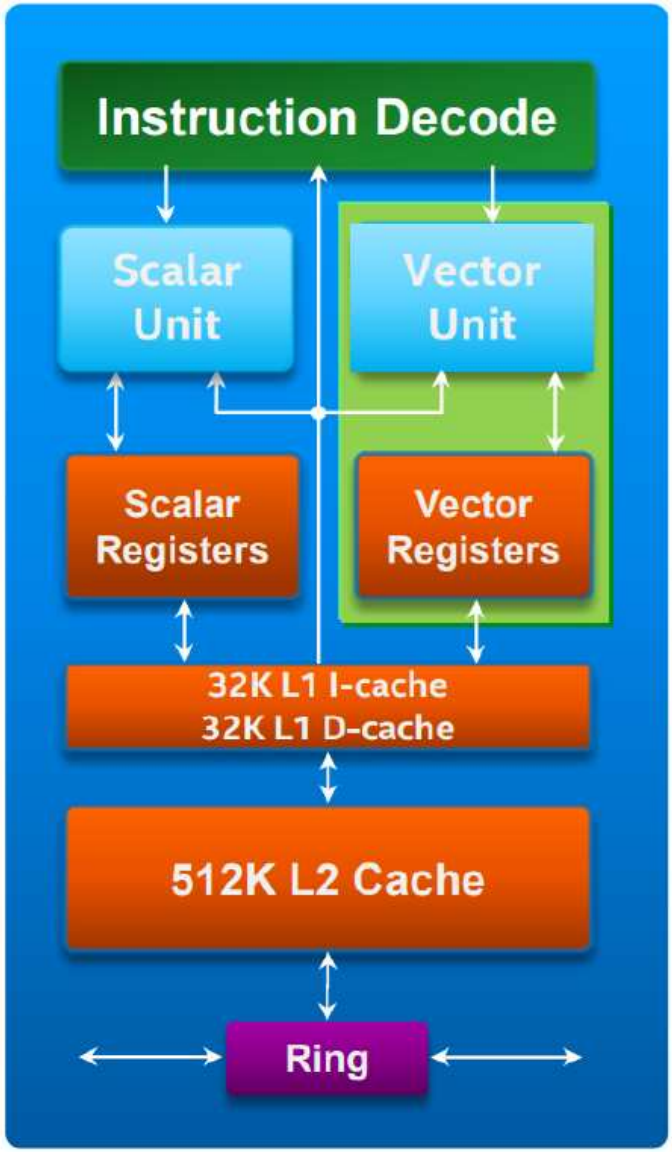
MIC Xeon Phi Ring



https://newsroom.intel.com/wp-content/uploads/sites/11/2016/01/Intel_Xeon_Phi_Hotchips_architecture_presentation.pdf

Core architecture

- Fetches and decodes instructions from four hardware thread execution contexts
  - The execution has a 4 clock latency, hidden by round-robin scheduling of threads
- Executes the x86 ISA, and Knights Corner vector instructions
- The core can execute 2 instructions per clock cycle, one per pipe
- SIMD vector processing engine 32 kB, 8-way set associative L1 I-cache and D-cache
- Coherent 512 kB L2 cache per core



https://software.intel.com/sites/default/files/Intel%C2%AE_Xeon_Phi%E2%84%A2_Coprocessor_Architecture_Overview.pdf

Comparison with a multicore CPU

Multi-Core Architecture

- C/C++/Fortran; OpenMP/MPI
- Standard Linux OS
- Up to 768 GB of DDR3 RAM
- $\leq$ 18 cores $\approx$ 3 GHz
- 2-way hyper-threading
- 256-bit AVX vectors

Intel Many Integrated Core (MIC) Architecture

- C/C++/Fortran; OpenMP/MPI
- Special Linux uOS distribution
- 6-16GB cached GDDR5 RAM
- 57-61 cores at $\approx$ 1GHz
- 4 hardware threads per core
- 512-bit IMCI vectors

Comparison with a CPU

- CPUs are designed for all workloads, high single-thread performance
- MIC is optimized for number crunching
  - Focus on high aggregate throughput via lots of weaker threads regularly achieves $> 2\times$ performance of dual E5 CPUs

	MIC (SE10P)	CPU (E5)	MIC is...
Number of cores	61	8	much higher
Clock speed (GHz)	1	2.7	lower
SIMD width (bits)	512	256	higher
DP GLFLOPS/core	16+	21+	lower
HO threads/core	4	1	higher

Comparison with a GPU with CUDA

- Both Xeon Phi and GPUs can be used as accelerators:
  - Accelerate applications in *offload mode*, where portions of the application are accelerated by a remote device
  - Run many threads in parallel to achieve massive parallelism
- Xeon Phi is not just an accelerator:
  - In *coprocessor native execution mode*, the Phi appears as another machine connected to the host, like another node in a cluster
  - In *symmetric execution mode*, application processes run on both the host and the coprocessor, communication through message passing
  - MIMD vs SIMD:
    - ★ CUDA threads are grouped in warps (work groups in OpenCL) in a SIMD (single instruction, multiple data) model
    - ★ Phi coprocessors run generic MIMD threads individually in a MIMD (multiple instruction, multiple data) model

Summary of MIC advantages

- Programming MIC is similar to programming for CPUs
  - Code compatible (C, C++, Fortran) with re-compilation
  - Any code can run on MIC, not just kernels
- Offers a new, flexible offload programming paradigm
  - C/Fortran markup to denote code to execute on Phi at runtme
  - Link to maths kernel library (MKL) implementation, which can offload automatically
- Goal: Optimizing for MIC is similar to optimizing for CPUs
  - Optimize once, run anywhere

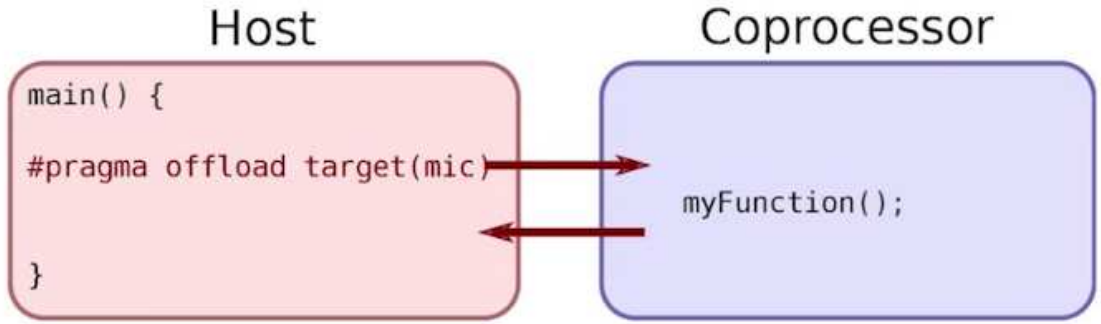
Drawback

- In practice, just as much recoding as for GPUs
- The future of Xeon Phi at Intel is unclear
  - (It doesn't have the economy-of-scale from the gamer market.)

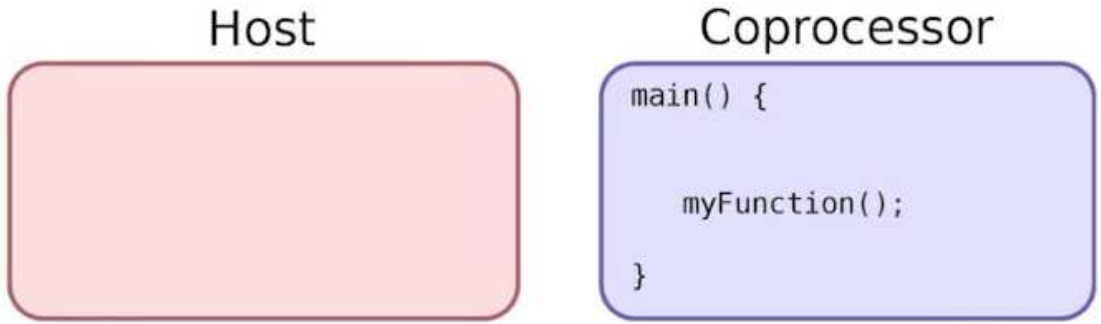
Models for Xeon Phi programming

Offload and Native modes

- Offload mode (explicit)

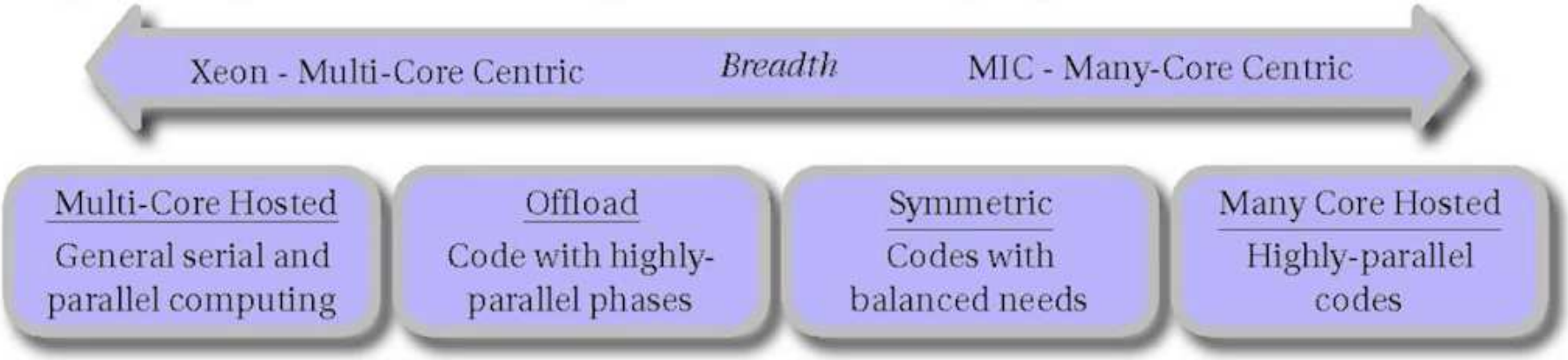


- Native mode (standalone application/ MPI process):



Coupling modes

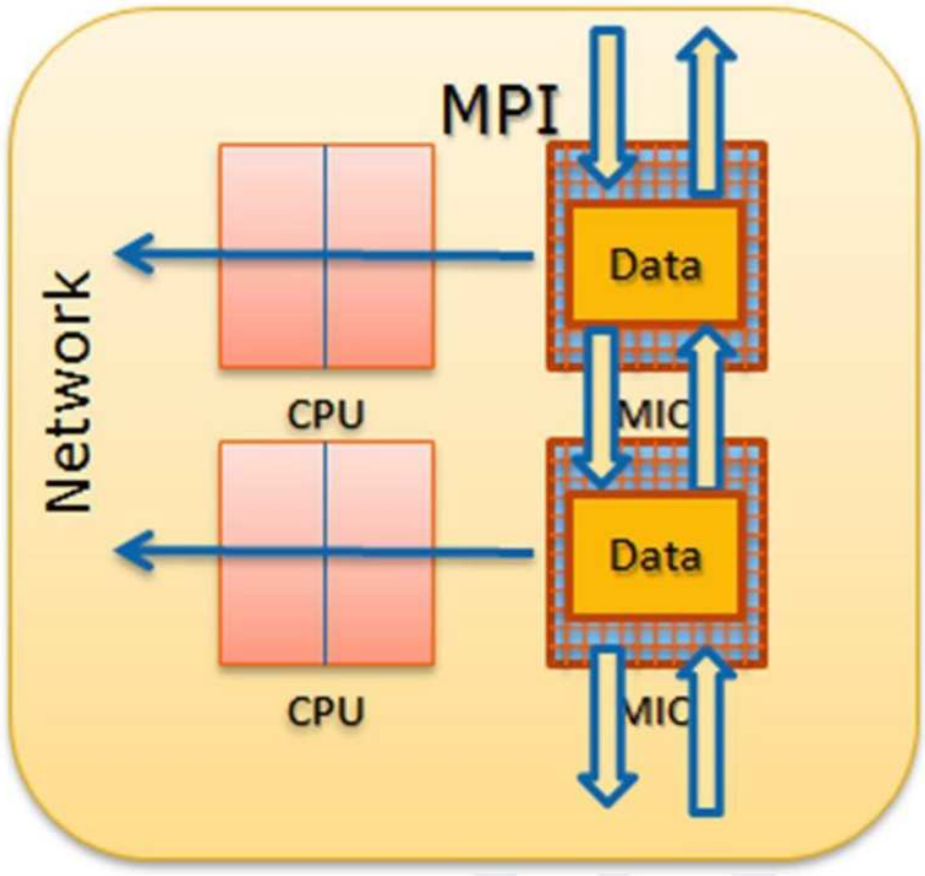
Programming models allow a range of CPU+MIC coupling modes



Native Execution

Native execution

- The MPI/openMP processes reside on the MIC coprocessor only
- MPI/openMP libraries, the application, and other needed libraries are uploaded to the coprocessors



Native Execution

"Hello World" application

```
#include <stdio.h>
#include <unistd.h>

int main() {
  printf("Hello world! I have %ld logical cores.\n",
  sysconf(_SC_NPROCESSORS_ONLN));

}
```

```
icc hello.c
./a.out
```

Hello world! I have 16 logical cores.

Native Execution

Compile and run the same code on the coprocessor in the native mode: The tool micnativeloadex automatically transfers code and dependent libraries and runs the application.

```
export SINK_LD_LIBRARY_PATH=/usr/local/intel/
        composer_xe_2015/lib/mic/
icc hello.c -mmic
micnativeloadex a.out
```

Hello world! I have 240 logical cores.

- Use -mmic to produce executable for MIC architecture
- Set `SINK_LD_LIBRARY_PATH` to help the tool find libraries
- Runs under micuser on coprocessor

Matrix-Matrix Multiplication — Native Mode

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

int main (int argc, char **argv)
{
    int n ;
    int i, j, k ;
    double time1, time2 ;
    # pragma omp parallel
    {
        int nthreads, threadid ;
        nthreads = omp_get_num_threads ();
        threadid = omp_get_thread_num ();
        if (threadid == 0) {
            printf ("Open-MP, threads = %d\n", nthreads);
        }
    }
```

Matrix-Matrix Multiplication — Native Mode

```
if (argc != 2) {
    fprintf(stderr,"Usage: %s matrix size\n", argv [0]
    exit (EXIT_FAILURE);
}
n = atoi (argv [1]);
if (n > 0) {
    printf ("Matrix size is %d\n",n);
}
else {
    fprintf (stderr, "Error, matrix size is %d\n", n);
    exit (EXIT_FAILURE);
}
double (*a)[n] = malloc (sizeof (double [n][n]));
double (*b)[n] = malloc (sizeof (double [n][n]));
double (*c)[n] = malloc (sizeof (double [n][n]));
```

Matrix-Matrix Multiplication — Native Mode

```
if (a == NULL || b == NULL || c == NULL) {
fprintf (stderr, "Not enough memory!\n");
    exit (EXIT_FAILURE);
}
for (i =0; i < n ; i ++)
    for (j =0; j < n ; j ++) {
        a [i][j] = ((double)rand())/((double)RAND_MAX);
        b [i][j] = ((double)rand())/((double)RAND_MAX);
        c [i][j] = 0.0;
    }
time1 = omp_get_wtime ();
# pragma omp parallel for private (k, j)
for (i =0; i < n ; i ++)
    for (k =0; k < n ; k ++)
        for (j =0; j < n ; j ++) {
            c [i][j] += a [i][k]* b [k][j];
        }
time2 = omp_get_wtime () - time1 ;
```

Matrix-Matrix Multiplication — Native Mode

```
    // check a random element
    i = rand ()% n ;
    j = rand ()% n ;
    double d = 0.0;
    for (k =0; k < n ; k ++)
        d += a [i][k]* b [k][j];

    printf ("Chk␣rnd␣element:␣%18.9lE\n", fabs (d -
    printf ("Elapsed␣time (s)␣=␣%f\n", time2;
    return 0;
}
```

Running Native OpenMP Applications

Source code: `shared/xeonphi/hello_openmp.c`

To compile OpenMP program for native execution:

```
export SINK_LD_LIBRARY_PATH=/usr/local/intel/
       composer_xe_2015/lib/mic/
icc -openmp -mmic hello_openmp.c
```

Running Native OpenMP Applications

script:

```
#!/bin/bash
#SBATCH --time=00:01:00
#SBATCH --nodes=1
#SBATCH --gres=mic

echo 'Job number'
echo $SLURM_CPUS_ON_NODE

micnativeloadex a.out -a 100
```

To request a single Xeon Phi

Explicit Offload

Offload Execution

- In this mode, MPI communication takes place only between host processors
- The co-processors are used only through the offload capabilities

Offload Execution

- Option 1: With compiler-assisted offload, you write code with offload annocations
  - No specific compiler flags needed, ofload is implicit where markup is encountered
  - Offload code will automatically run on MIC at runtime if MIC is present, otherwise a host version is run.
- Option 2: With authomatic offload, you link to a library that can perform offload operations (e.g., MKL, Math Kernel Library)
  - MKL is offload-capable; all you do is link to it (`-lmkl`)
  - Need to explicitly tell MKL to use offload at runtime via environment variable `MKL_MIC_ENABLE=1`

Offload: Pragma-based approach

"Hello World" in the explicit offload model:

```
#include <stdio.h>

int main(int argc, char * argv[]) {
  printf("Hello␣World␣from␣host!\n");

#pragma offload target(mic)
{
  printf("Hello␣World␣from␣coprocessors!\n");
}
  printf("Bye\n");
}
```

Application runs on the host, but some parts of the code and data are offloaded to the coprocessor.

Offload: Pragma-based approach

Important: set environment variables

```
source /usr/local/intel/composerxe/bin/
        compilervars.sh intel64
```

Request a single Xeon Phi coprocessor

```
# SBATCH --gres = mic
```

Offload: Pragma-based approach

To compile OpenMP program for offload execution

```
icc hello_offload.c -o hello_offload
./hello_offload
```

- No additional arguments if compiled with an Intel compiler
- Run application on host as a regular application
- Code inside of `#pragmaoffload` is offloaded automatically
- Console output on Intel Xeon Phi coprocessor is buffered and mirrored to the host console
- If coprocessor is not installed, code inside `#pragramoffload` runs on the host system

Offload Functions

```
__attribute__((target(mic))) void MyFunction() {
  // .. implement function as usual
}

int main(int argc, char *argv[]) {

#pragma offload target(mic)
{
  MyFunction;
}
}
```

- Functions used on coprocessor must be marked with the specifier `__attribute__((target(mic)))`
- Compiler produces a host version and a coprocessor version of such functions(to enable fall-back to host)

Offload Multiple Functions

```
#pragma offload_attribute(push, target(mic))
void MyFunctionOne() {
// ... implement function as usual
}

void MyFunctionTwo() {
// ... implement function as usual
}
#pragma offload_attribute(pop)
```

- To mark a long block of code with the offload attribute, use #pragmaoffload_attribute(push/pop)

Matrix-Matrix Multiplication — Offload Mode

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

void MatMatMult (int size,
    float (* restrict A)[size],
    float (* restrict B)[size],
    float (* restrict C)[size])
{
    # pragma offload target (mic) \
    in (A : length (size * size)) \
    in (B : length (size * size)) \
    out (C : length (size * size))
    {
```

Matrix-Matrix Multiplication — Offload Mode

```
        # pragma omp parallel for \
                default (none) shared (C, size)
        for (int i =0; i < size; i ++)
            for (int j =0; j < size; j ++)
                C [i][j] = 0.0 f;
        # pragma omp parallel for default (none) shared (A
        for (int i =0; i < size; i ++)
          for (int j =0; j < size; j ++)
              for (int k =0; k < size; k ++)
                  C [i][j] += A [i][k] * B [k][j];
    }
}
```

Matrix-Matrix Multiplication — Offload Mode

```
int main (int argc, char * argv [])
{
    if (argc != 4) {
        fprintf (stderr, "Use␣:␣%␣s␣size␣nThreads␣nIter\n"
        return -1;
    }
    int i, j, k;
    int size = atoi (argv [1]);
    int nThreads = atoi (argv [2]);
    int nIter = atoi (argv [3]);
    omp_set_num_threads (nThreads);
    float (*restrict A)[size] = malloc(sizeof (float)* si
    float (*restrict B)[size] = malloc(sizeof (float)* si
    float (*restrict C)[size] = malloc(sizeof (float)* si
```

Matrix-Matrix Multiplication — Offload Mode

```
# pragma omp parallel for default (none) \
          shared (A,B, size) private (i,j, k)
for (i =0; i < size; i ++)
    for (j =0; j < size; j ++)
    {
        A [i][j] = (float) i + j;
        B [i][j] = (float) i - j;
    }
double avgMultTime = 0.0;
MatMatMult (size, A, B, C); // warm up
double startTime = dsecnd ();
for (int i=0; i < nIter; i ++)
     MatMatMult (size, A, B, C);
double endTime = dsecnd ();

avgMultTime = (endTime - startTime)/ nIter;
```

Matrix-Matrix Multiplication — Offload Mode

```
    # pragma omp parallel
    # pragma omp master
    printf("%s␣nThrds␣%d␣mx␣%d␣%d␣time␣%g␣GFlop/s␣%g\n",
        argv [0], omp_get_num_threads (), size, size,
        avgMultTime, 2e-9*size*size*siz  avgMultTime);

    # pragma omp barrier
    free (A); free (B); free (C);
    return 0;
}
```

Reference

http://www.colfax-intl.com/nd/resources/slides.aspx
http://www.prace-ri.eu/best-practice-guide-intel-xeon-phi-html/