MIC: Introduction to Xeon Phi and Symmetric Computing

Presenter: Steve Lantz, CAC

Coauthors: Lars Koesterke and John Cazes, TACC

Xeon Phi — MIC

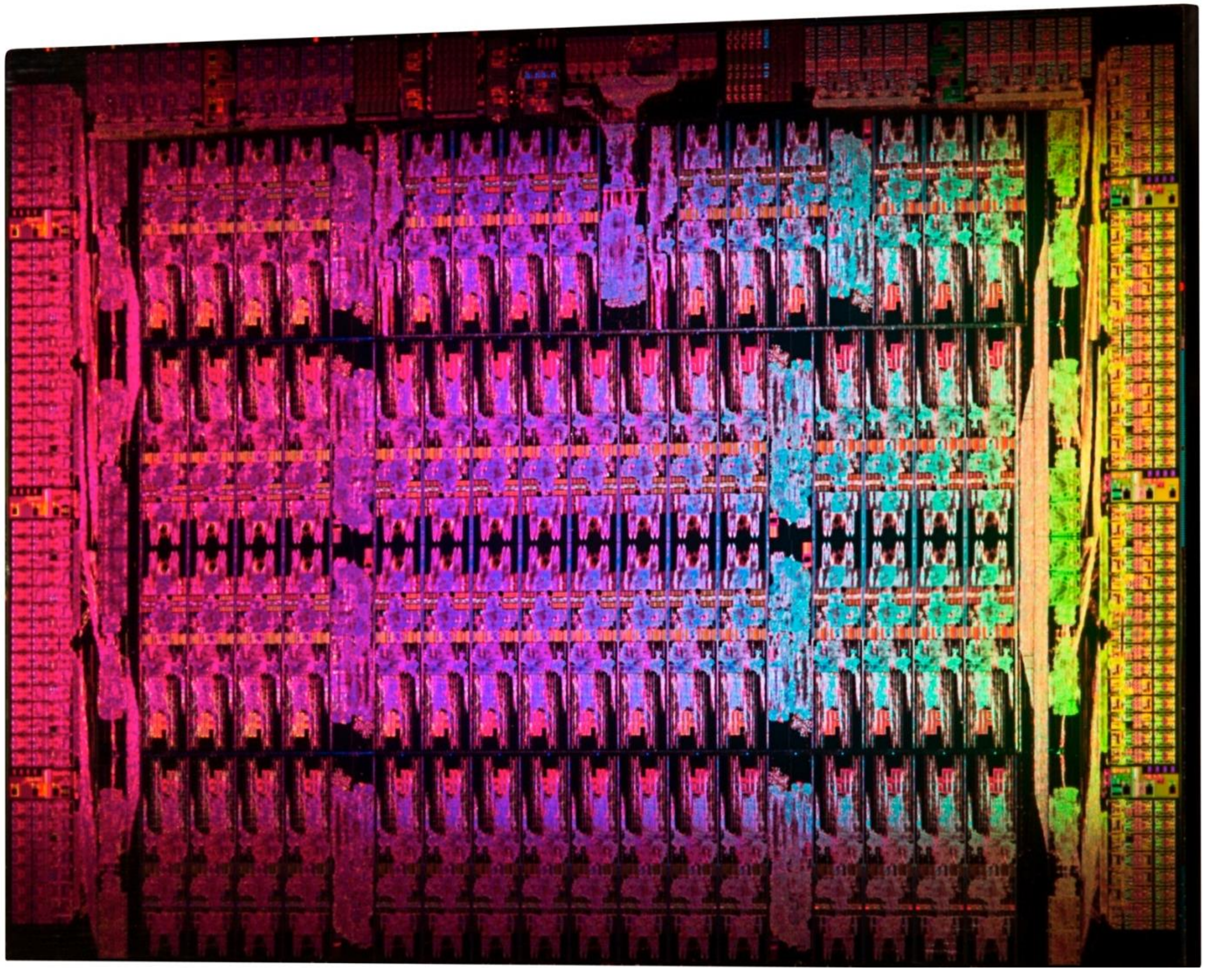
- Xeon Phi = first product of Intel's Many Integrated Core (MIC) architecture
- Co-processor
  - PCI Express card
  - Stripped down Linux operating system
- Dense, simplified processor
  - Many power-hungry operations removed
  - Wider vector unit
  - Wider hardware thread count
- Lots of names
  - Many Integrated Core architecture, aka MIC
  - Knights Corner (code name)
  - Intel Xeon Phi Co-processor SE10P (product name)

Xeon Phi — MIC

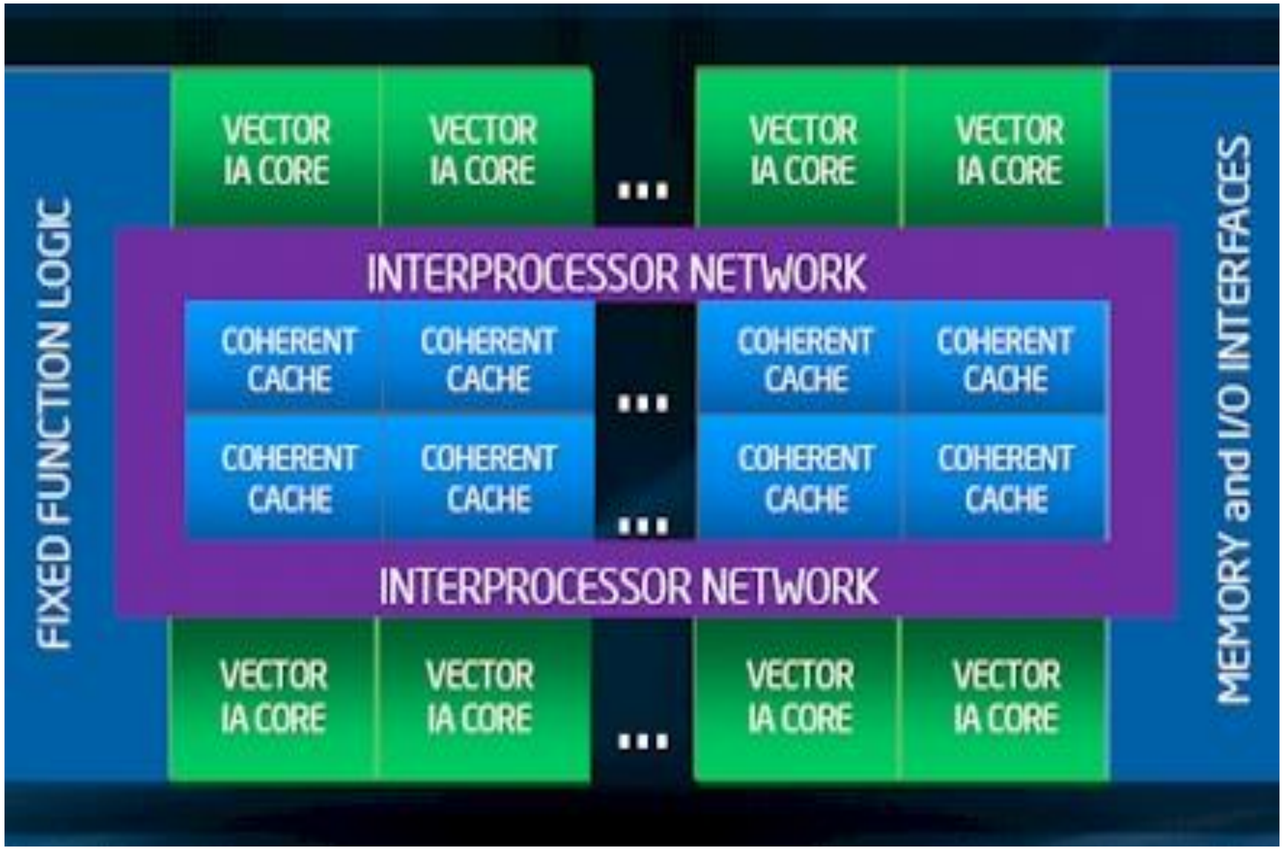
- Leverage x86 architecture (CPU with many cores)
  - x86 cores that are simpler, but allow for more compute throughput
- Leverage existing x86 programming models
- Dedicate much of the silicon to floating point ops
- Cache coherent
- Increase floating-point throughput
- Strip expensive features
  - out-of-order execution
  - branch prediction
- Widen SIMD registers for more throughput
- Fast (GDDR5) memory on card

Intel Xeon Phi Chip

- 22 nm process
- Based on what Intel learned from
  - Larrabee
  - SCC
  - TeraFlops Research Chip

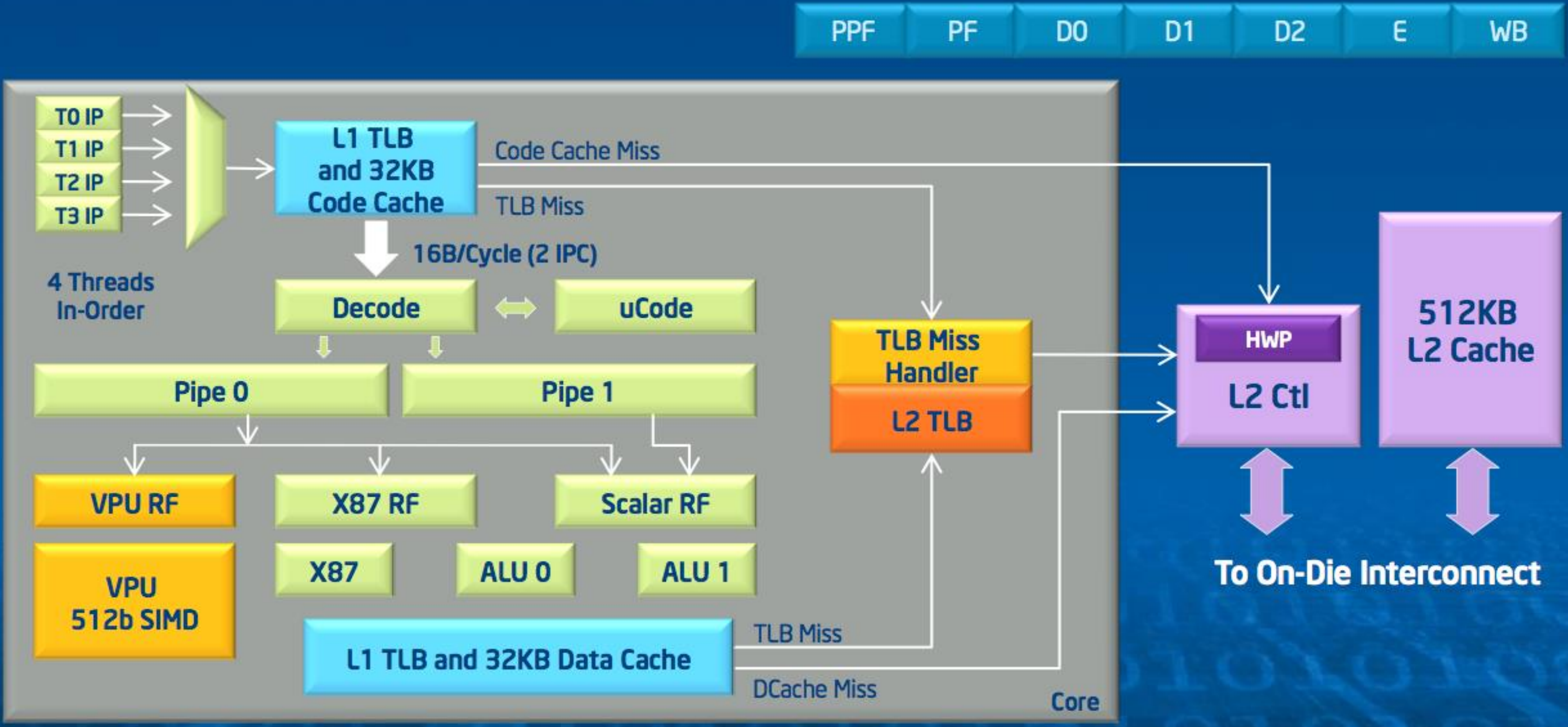


MIC Architecture



- Many cores on the die
- L1 and L2 cache
- Bidirectional ring network for L2
- Memory and PCIe connection

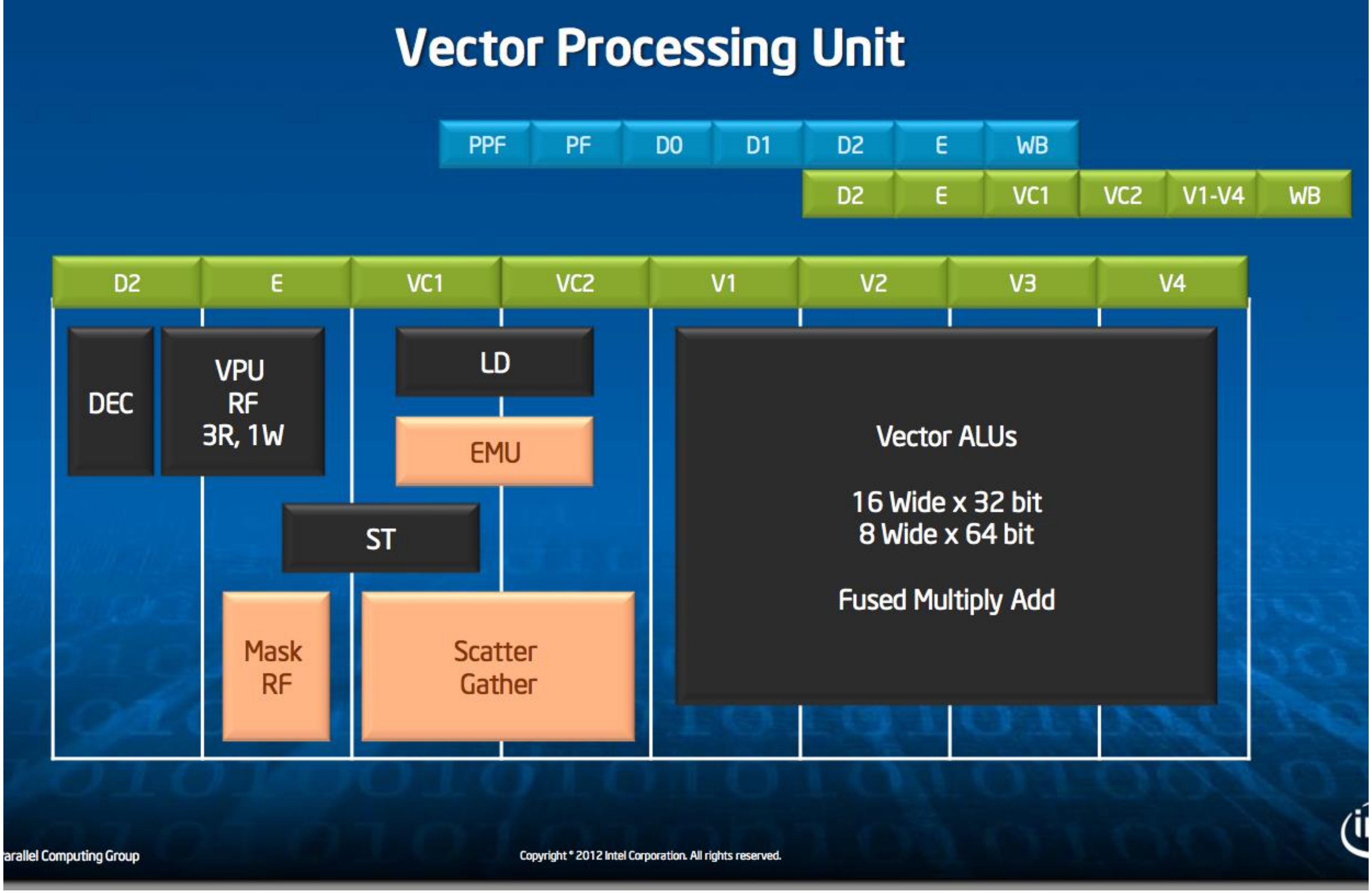
Knights Corner Core



X86 specific logic < 2% of core + L2 area

Visual and Parallel Computing Group

George Chrysos, Intel, Hot Chips 24 (2012):
http://www.slideshare.net/IntelXeon/under-the-armor-of-knights-corner-intel-mic-architecture-at-hotchips-2012



Vector Processing Unit

George Chrysos, Intel, Hot Chips 24 (2012):
http://www.slideshare.net/IntelXeon/under-the-armor-of-knights-corner-intel-mic-architecture-at-hotchips-2012

Speeds and Feeds

- Processor
  - ~1.1 GHz
  - 61 cores
  - 512-bit wide vector unit
  - 1.074 TF peak DP
- Data Cache
  - L1 32KB/core
  - L2 512KB/core, 30.5 MB/chip
- Memory
  - 8GB GDDR5 DRAM
  - 5.5 GT/s, 512-bit*
- PCIe
  - 5.0 GT/s, 16-bit

Stampede Programming Models

- Traditional Cluster
  - Pure MPI and MPI+X
    - X may be OpenMP, TBB, Cilk+, OpenCL, …
- Native Phi
  - Use one Phi and run OpenMP or MPI programs directly
- MPI tasks on Host and Phi
  - Treat the Phi (mostly) like another host
    - Pure MPI and MPI+X (limited memory: using 'X' is almost mandatory)
- MPI on Host, Offload to Xeon Phi
  - Targeted offload through OpenMP extensions
  - Automatically offload some library routines with MKL

Traditional Cluster

- Stampede is 2+ PF of FDR-connected Xeon E5
  - High bandwidth: 56 Gb/s (sustaining >52 Gb/s)
  - Low-latency
    - ~1 μs on leaf switch
    - ~2.5 μs across the system
- Highly scalable for existing MPI codes
- IB multicast and collective offloads for improved collective performance

Native Execution

- Build for Phi with -mmic
- Execute on host (runtime will automatically detect an executable built for Phi)
- … or ssh to mic0 and run on the Phi
- Can safely use all 61 cores
  - But: I recommend to use 60 cores, i.e. 60, 120, 180, or 240 threads
  - Offload programs should **certainly** stay away from the 61st core since the offload daemon runs here

Symmetric MPI

- Host and Phi can operate symmetrically as MPI targets
  - High code reuse
  - MPI and hybrid MPI+X (X = OpenMP, Cilk+, TBB, pthreads)
- Careful to balance workload between big cores and little cores
- Careful to create locality between local host, local Phi, remote hosts, and remote Phis
- Take advantage of topology-aware MPI interface under development in MVAPICH2
  - NSF STCI project with OSU, TACC, and SDSC

Symmetric MPI

- Typical 1-2 GB per task on the host
- Targeting 1-10 MPI tasks per Phi on Stampede
  - With 6+ threads per MPI task
  - Still 1-2 GB per task, but not per thread

MPI with Offload to Phi

- Existing codes using accelerators have already identified regions where offload works well
- Porting these to OpenMP offload should be straightforward
- Automatic offload where MKL kernel routines can be used
  - xGEMM, etc.

MPI with Offload Sections

ADVANTAGES

- Offload Sections may easily be added to hybrid MPI/OpenMP codes with directives
- Intel compiler will automatically detect and compile offloaded sections

CAVEATS

- No MPI calls are allowed within offload sections
- Each host task may spawn an offload section

Summary: Advantages of MIC

- Intel's MIC is based on x86 technology
  - x86 cores w/ caches and cache coherency
  - SIMD instruction set
- Programming for Phi is similar to programming for CPUs
  - Familiar languages: C/C++ and Fortran
  - Familiar parallel programming models: OpenMP & MPI
  - MPI on host and on the coprocessor
  - Any code can run on MIC, not just kernels
- Optimizing for Phi is similar to optimizing for CPUs
  - **"Optimize once, run anywhere"**
  - Early Phi porting efforts for codes "in the field" have obtained double the performance of Sandy Bridge

Will My Code Run on Xeon Phi?

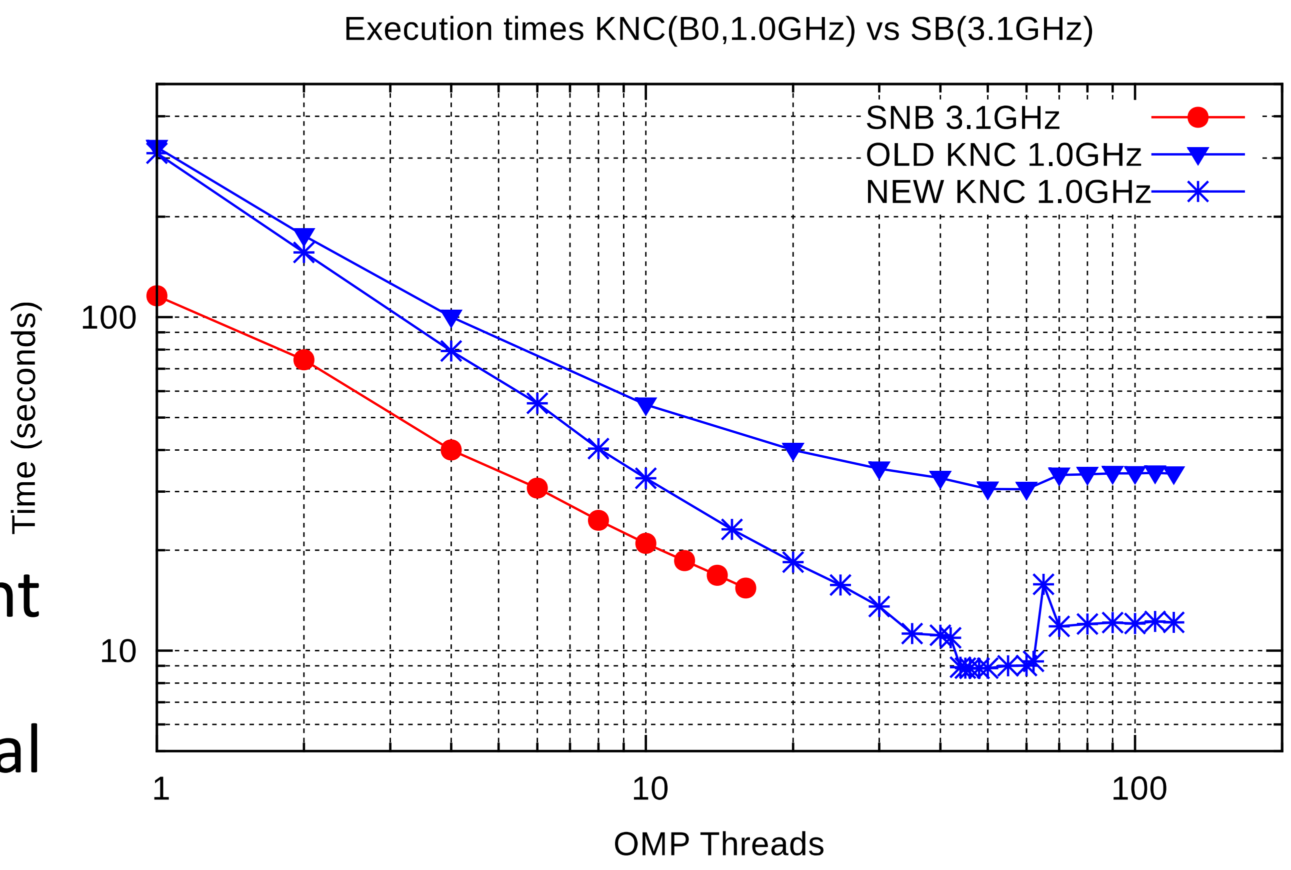
- Yes
- … but that's the wrong question
  - Will your code run *best* on Phi?, or
  - Will you get great Phi performance without additional work? (The answer is most likely **NO**)

Early Phi Programming Experiences at TACC

- Codes port easily
  - Minutes to days depending mostly on library dependencies
- Performance can require real work
  - While the software environment continues to evolve
  - Getting codes to run *at all* is almost too easy; really need to put in the effort to get what you expect
- Scalability is pretty good
  - Multiple threads per core is really important
  - Getting your code to vectorize is really important

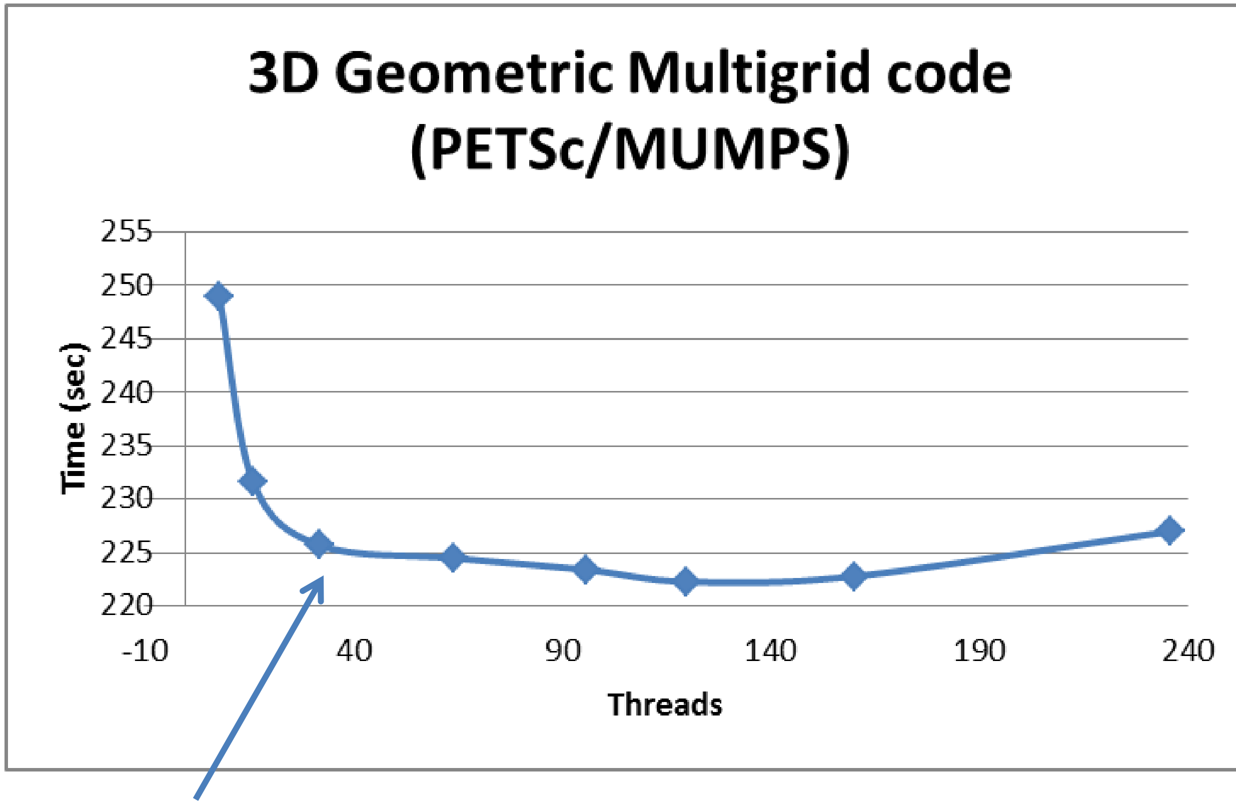
LBM Example: Native on Phi vs. Host

- Lattice Boltzmann Method CFD code
  - Carlos Rosales, TACC
  - MPI code with OpenMP
- Finding all the right routines to parallelize is critical



PETSc/MUMPS with AO

- Hydrostatic ice sheet modeling
- MUMPS solver (called through PETSc)
- BLAS calls to MKL automatically offloaded behind the scenes*



*Increasing threads doesn't always help!

More on Symmetric Computing

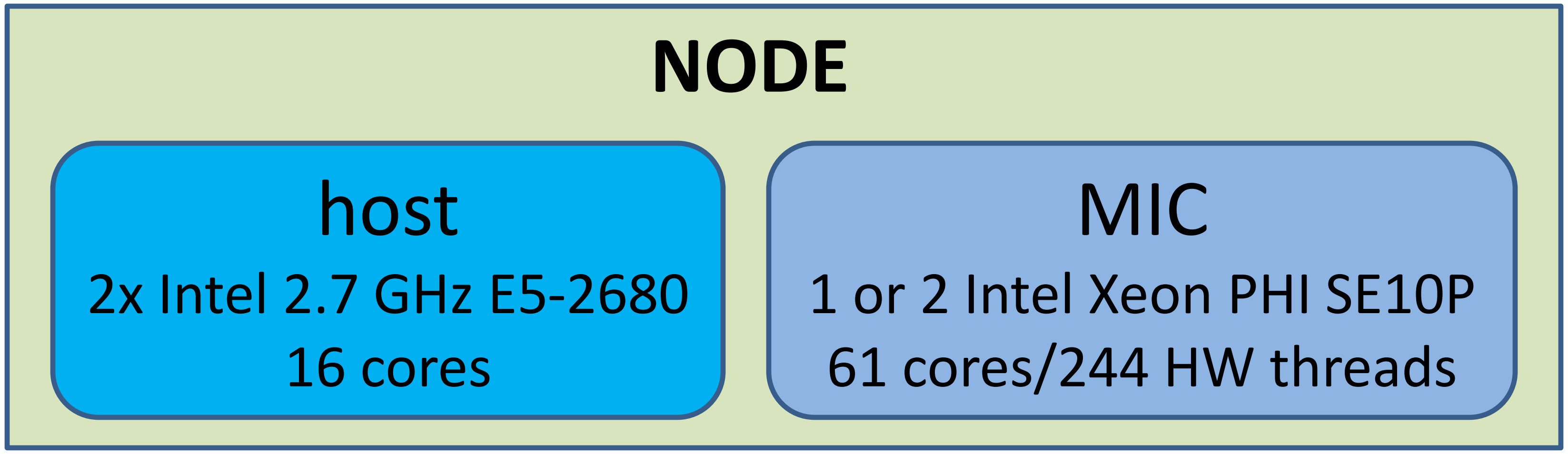
Run MPI tasks on both MIC and host and across nodes

- Also called “heterogeneous computing”
- Two executables are required:
  - CPU
  - MIC
- Currently only works with Intel MPI
- MVAPICH2 support coming

Definition of a Node

A "node" contains a host component and a MIC component

- host – refers to the Sandy Bridge component
- MIC – refers to one or two Intel Xeon Phi co-processor cards



Environment Variables for MIC

By default, environment variables are "inherited" by all MPI tasks

Since the MIC has a different architecture, several environment variables must be modified

- LD_LIBRARY_PATH – must point to MIC libraries
- I_MPI_PIN_MODE – controls the placement of tasks
- OMP_NUM_THREADS – # of threads on MIC
- KMP_AFFINITY – controls thread binding

Steps to Create a Symmetric Run

1. Compile a host executable and a MIC executable:
   – mpicc –openmp –o my_exe.cpu my_code.c
   – mpicc –openmp –mmic –o my_exe.mic my_code.c
2. Determine the appropriate number of tasks and threads for both MIC and host:
   – 16 tasks/host – 1 thread/MPI task
   – 4 tasks/MIC – 30 threads/MPI task

Steps to Create a Symmetric Run

3. Create a batch script to distribute the job

```
#!/bin/bash
#-----------------------------------------------------
# symmetric.slurm
# Generic symmetric script – MPI + OpenMP
#-----------------------------------------------------
#SBATCH -J symmetric        # Job name
#SBATCH -o symmetric.%j.out # stdout; %j expands to jobid
#SBATCH -e symmetric.%j.err # stderr; skip to combine stdout and stderr
#SBATCH -p development      # queue
#SBATCH -N 2                # Number of nodes, not cores (16 cores/node)
#SBATCH -n 32               # Total number of MPI tasks (if omitted, n=N)
#SBATCH -t 00:30:00         # max time
#SBATCH -A TG-01234         # necessary if you have multiple projects

export MIC_PPN=4
export MIC_OMP_NUM_THREADS=30

ibrun.symm -m ./my_exe.mic –c ./my_exe.cpu
```

Symmetric Launcher – ibrun.symm

Usage:

```
ibrun.symm  -m ./<mic_executable>  -c ./<cpu_executable>
```

- Analog of ibrun for symmetric execution
- # of MIC tasks and threads are controlled by env variables

MIC_PPN = <# of MPI tasks/MIC card>

MIC_OMP_NUM_THREADS = <# of OMP threads/MIC MPI task>

MIC_MY_NSLOTS = < Total # of MIC MPI tasks >

Symmetric Launcher

- # of host tasks determined by batch script (same as regular ibrun)
- ibrun.symm does not support "-o" and "-n" flags
- Command line arguments may be passed with quotes

```
ibrun.symm -m "./my_exe.mic args" -c "./my_exe.cpu args"
```

Symmetric Launcher

- If the executables require redirection or complicated command lines, a simple shell script may be used:

run_mic.sh	run_cpu.sh

`#!/bin/sh`
`a.out.mic <args> < inputfile`
	`#!/bin/sh`
	`a.out.host <args> < inputfile`

```
ibrun.symm -m ./run_mic.sh -c ./run_cpu.sh
```

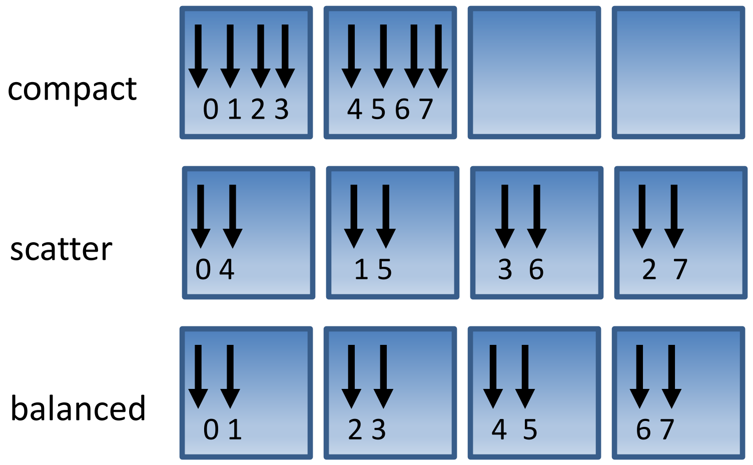
Note: The bash, csh, and tcsh shells are not available on MIC.
So, the MIC script must begin with "`#!/bin/sh`"

Thread Placement

Thread placement may be controlled with the following environment variable

- KMP_AFFINITY=<type>

compact	pack threads close to each other
scatter	Round-Robin threads to cores
balanced	keep OMP thread ids consecutive (MIC only)
explicit	use the proclist modifier to pin threads
none	does not pin threads



KMP_AFFINITY=balanced (Default for ibrun.symm)

Balance

- How to balance the code?

	Sandy Bridge	**Xeon Phi**
Memory	32 GB	8 GB
Cores	16	61
Clock Speed	2.7 GHz	1.1 GHz
Memory Bandwidth	51.2 GB/s(x2)	352 GB/s
Vector Length	4 DP words	8 DP words

Balance

Example: Memory balance

Balance memory use and performance by using a different # of tasks/threads on host and MIC

Host
16 tasks/1 thread/task
2GB/task

Xeon PHI
4 tasks/60 threads/task
2GB/task

Balance – Lab Exercise

Example: Performance balance

Balance performance by tuning the # of tasks and threads on host and MIC

Host
? tasks/? threads/task
?GB/task

Xeon PHI
? tasks/? threads/task
?GB/task

MIC Offloading with OpenMP

- In OpenMP 4.0, accelerator syntax may ultimately be standardized,
- For now, we use special MIC directives for the Intel compilers
- OpenMP pragma is preceded by MIC-specific `pragma`
  - Fortran: `!dir$ omp offload target(mic) <...>`
  - C: `#pragma offload target(mic) <...>`
- All data transfer is handled by the compiler
  - User control provided through `optional keywords`
- I/O can be done from within offloaded region
  - Data can “stream” to the MIC; no need to leave MIC to fetch new data
  - Can be very helpful when debugging (just insert print statements)
- Specific subroutines can be offloaded, including MKL subroutines

MIC Example 1

2-D array (**a**) is filled with data on the coprocessor

Data management done automatically by compiler

- Memory is allocated on coprocessor for (**a**)
- Private variables (**x,i,j**) are created
- Result is copied back

```
use omp_lib                              ! Fortran example
integer                ::  n = 1024        ! Size
real(:,:), allocatable ::  a               ! Array
integer                ::  i, j            ! Index
real                   ::  x               ! Scalar
allocate(a(n,n))                         ! Allocation
!dir$ omp offload target(mic)            ! Offloading
!$omp parallel shared(a,n), private(x)
!$omp do private(i,j), schedule(dynamic)
do j=1,n
  do i=j,n
    x = real(i + j); a(i,j) = x
```

```
#include <omp.h>        /* C example */
  const int n = 1024; /* Size of the array */
  int      i, j;      /* Index variables   */
  float a[n][n], x
#pragma offload target(mic)
#pragma omp parallel shared(a,n), private(x)
#pragma omp for private(i,j), schedule(dynamic)
  for(i=0;i<n;i++) {
    for(j=i;j<n;j++) {
      x = (float)(i + j); a[i][j] = x; }}
```

MIC Example 2

I/O from offloaded region:

- File is opened and closed by one thread (`omp single`)
- All threads take turns reading from the file (`omp critical`)

Threads may also read in parallel (not shown)

- Parallel file system
- Threads read parts from different targets

```
#pragma offload target(mic) //Offload region
#pragma omp parallel
{
  #pragma omp single /* Open File */
  {
  printf("Opening file in offload region\n");
  f1 = fopen("/var/tmp/mydata/list.dat","r");
  }

  #pragma omp for
  for(i=1;i<n;i++) {
    #pragma omp critical
    { fscanf(f1,"%f",&a[i]);}
    a[i] = sqrt(a[i]);
  }

  #pragma omp single
  {
  printf("Closing file in offload region\n");
  fclose (f1);
  }
}
```

MIC Example 3

Two routines, MKL's **sgemm** and **my_sgemm**

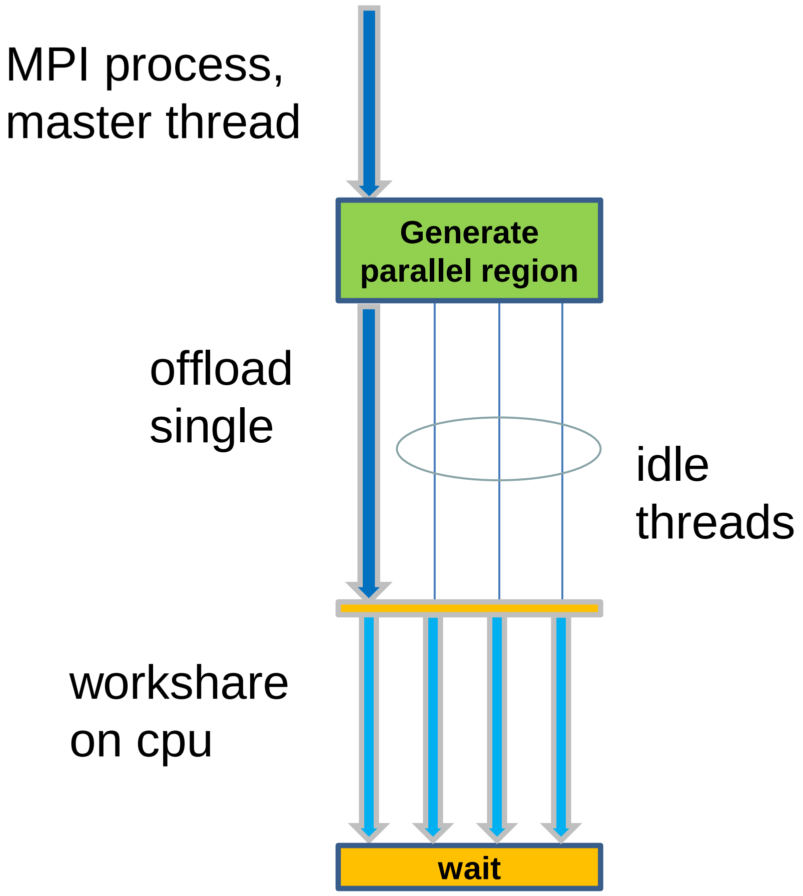
- Both are called with `offload` directive
- **my_sgemm** specifies explicit `in` and `out` data movement

Use `attributes` to have routine compiled for the coprocessor, or link coprocessor-based MKL

```
! snippet from the caller...
! offload MKL routine to accelerator
!dir$ attributes offload:mic :: sgemm
!dir$ omp offload target(mic)
call sgemm('N','N',n,n,n,alpha,a,n,b,n,beta,c,n)
! offload hand-coded routine with data clauses
!dir$ offload target(mic) in(a,b) out(d)
call my_sgemm(d,a,b)
```

```
! snippet from the hand-coded subprogram...
!dir$ attributes offload:mic :: my_sgemm
subroutine my_sgemm(d,a,b)
real, dimension(:,:) :: a, b, d
!$omp parallel do
do j=1,n
  do i=1,n
    d(i,j) = 0.0
    do k=1,n
      d(i,j) = d(i,j)+a(i,k)*b(k,j)
    enddo; enddo; enddo
end subroutine
```

Heterogeneous Threading, Sequential



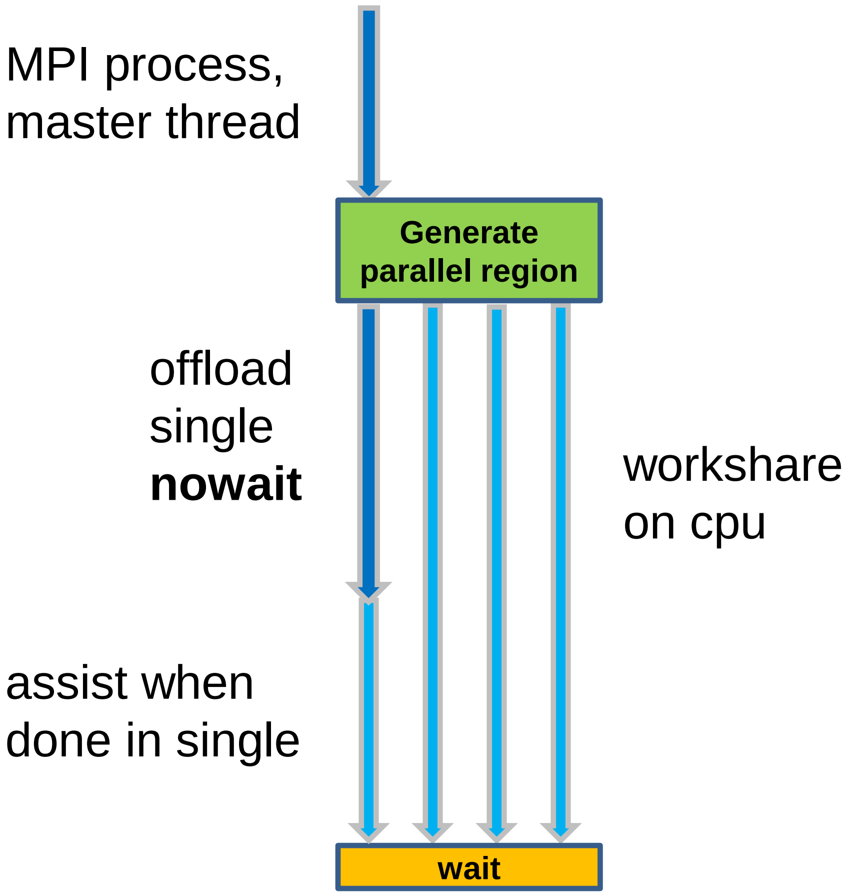
C/C++

```
#pragma omp parallel
 {
#pragma omp single
   { offload(); }

#pragma omp for
   for(i=0; i<N; i++){...}
 }
```

F90

```
!$omp parallel
   !$omp single
     call offload();
   !$omp end single

   !$omp do
     do i=1,N; ...
     end do
!$omp end parallel
```

Heterogeneous Threading, Concurrent



C/C++
```
#pragma omp parallel
 {
#pragma omp single nowait
   { offload(); }

#pragma omp for schedule(dynamic)
   for(i=0; i<N; i++){...}
 }
```

F90
```
!$omp parallel
   !$omp single
     call offload();
   !$omp end single nowait

   !$omp do schedule(dynamic)
     do i=1,N; ...
     end do
!$omp end parallel
```