Experience with a Prototype Intel® MIC System

Hamid Oloso amidu.o.oloso@nasa.gov

Kaushik Datta kaushik.datta@nasa.gov

Outline

- What is Intel's MIC Platform?
- MIC Hardware
- MIC Programming Models
- MIC SGEMM Performance
- FV 2D Advection Code
- Conclusion

Intel MIC Overview

- Intel MIC (Many Integrated Core)
  - A co-processor attached to CPU host via PCIe bus
  - Designed for highly parallel, vectorizable codes
  - Uses many small, simple cores with wide vector units
  - Getting full performance requires both a high degree of parallelism *and* vectorization
    - Not all code can be written this way
    - Not all programs make sense on this architecture
  - Prototype box named Knights Ferry (KNF) tested. Not a product. Used it to:
    - Prepare your code for the next-generation production hardware
    - Validate the programming models
    - Help us gather your feedback
  - First MIC consumer product (Knights Corner) expected 2012/2013

Outline

- What is Intel's MIC Platform?
- MIC Hardware
- MIC Programming Models
- MIC SGEMM Performance
- FV 2D Advection Code
- Conclusion

Our KNF Test Node

- 12 Xeon CPU cores (Westmere @ 3.33 GHz)
- 2 MIC co-processors (over PCIe) each with:
  - 30 active cores
  - High-BW bidirectional ring connecting cores (for cache coherency)
  - 2GB GDDR5 memory (graphics memory)
  - *Only 1 MIC co-processor used in our testing*

KNF MIC Core

- 4 hardware threads
- Two pipelines
  - Pentium® processor-based scalar units
  - 64-bit addressing
- 64 KB L1 cache and 256 KB L2 cache
  - Both are fully coherent
- All new vector unit
  - 512-bit SIMD instructions - not Intel® SSE, MMX™, or Intel® AVX
  - 32 512-bit wide vector registers
    - Hold 16 singles or 8 doubles per register
    - DP is currently much slower than SP
- Intel® MIC is not an Intel® Xeon® processor
  - It specializes in running highly parallel and vectorized code.
  - Not optimized for processing serial code

Outline

- What is Intel's MIC Platform?
- MIC Hardware
- MIC Programming Models
- MIC SGEMM Performance
- FV 2D Advection Code
- Conclusion

Programming Models

	native host (Xeon)	offload	symmetric	reverse offload	native MIC
Xeon	Program foo call bar End program	Program foo call bar End program	Program foo call bar End program	bar	
MIC		bar	Program foo call bar End program	Program foo call bar End program	Program foo call bar End program

- We only tested offload and native modes
- Used OpenMP + MIC directives for parallelization
- MPI should also be supported, but have not tested it

Outline

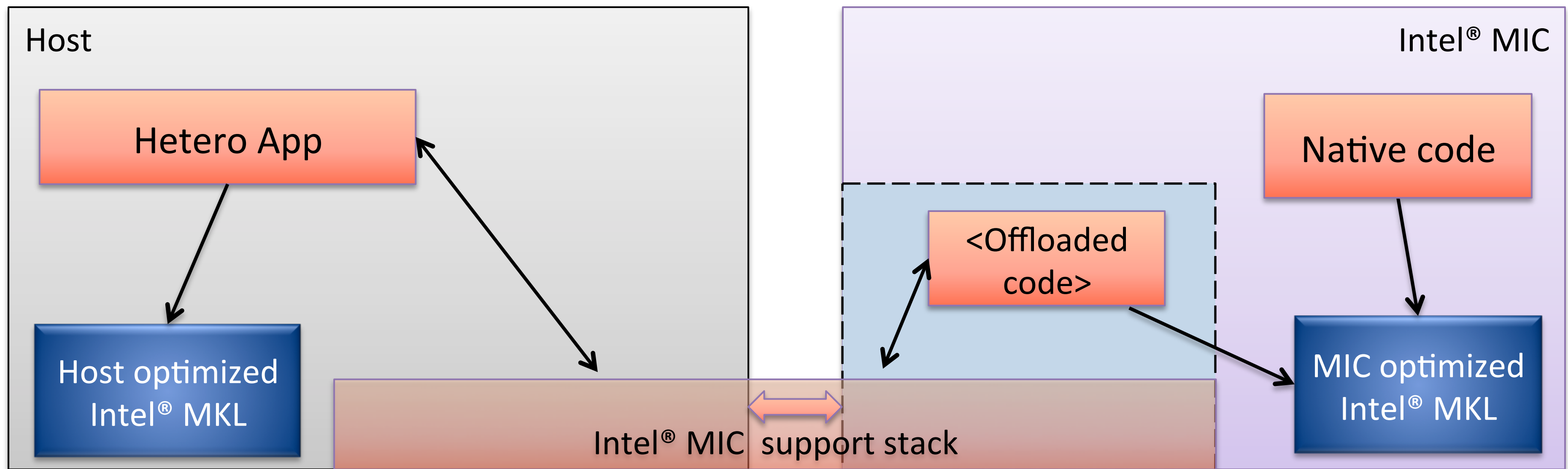
- What is Intel's MIC Platform?
- MIC Hardware
- MIC Programming Models
  - Offload Mode
  - Native Mode
- MIC SGEMM Performance
- FV 2D Advection Code
- Conclusion

Offload Mode Basics

- Need to specify what gets offloaded to MIC card:
  1. Data: `!dir$ omp offload target(mic:0) in(a) inout(b)`
  2. Subroutines: `!dir$ attributes offload:mic :: matmult`
- Offload directives must be followed by OpenMP parallel region
  - Distributes work over MIC cores
- Code compilation:
  - Requires special offload flags
    - -opt-subscript-in-range -align all **-offload-build -openmp** -O3
  - Generates report to show whether offloaded code is vectorized
    - **-vec-report3** -opt-subscript-in-range -align all **-offload-build -openmp** -O3
- Code execution:
  - Can control MIC parallelization via `OMP_NUM_THREADS`
  - Can control MIC affinity via `KMP_AFFINITY`
  - Can run MKL library calls on the MIC (as we will see…)

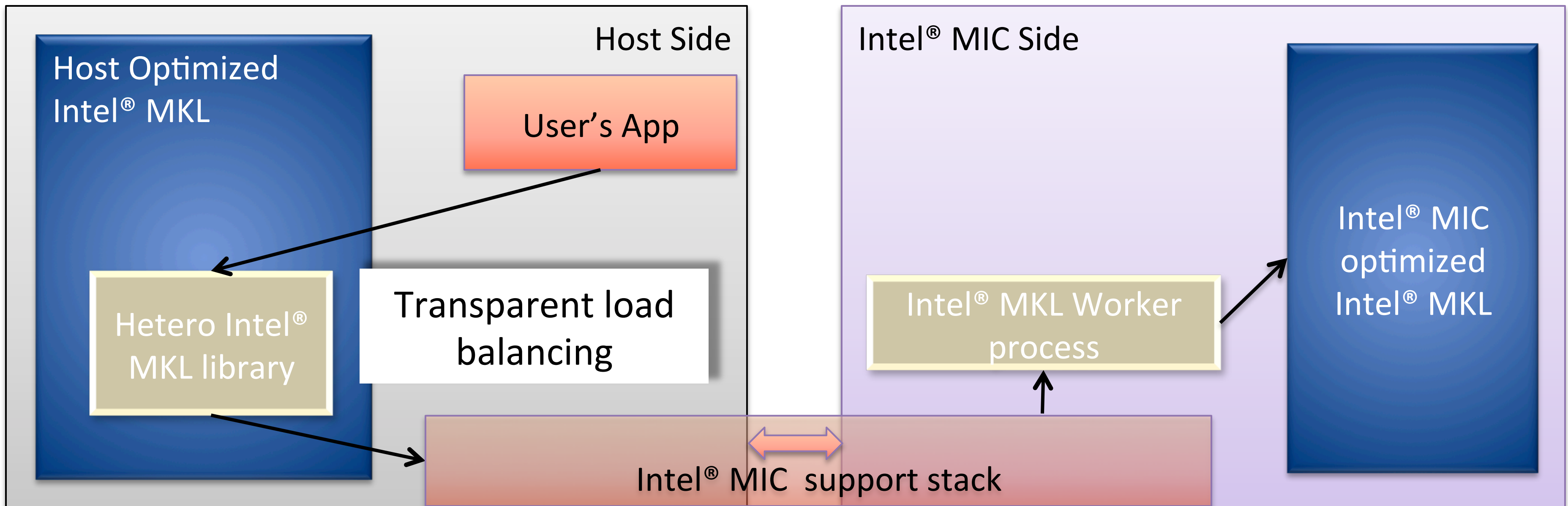
Intel® Math Kernel Library Use in Offload Code

- Native execution (of course)
- Identical usage syntax on host and coprocessor
- Functions called from the host execute on the host, functions called from the coprocessor execute on coprocessor
  - User is responsible for data transfer and execution management between the two domains



Intel® Math Kernel Library Automatic Offload

- Transparent load balancing between host and coprocessors
- Initiated by calling `mkl_mic_enable()` on the host before calling Intel® MKL functions that implement Automatic Offload
- Call the function from the host code
  - No "_Offload" or "#pragma offload" needed
  - Intel® MKL is responsible for data transfer and execution management



Offload Mode Caveats

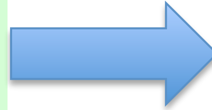
- Overheads:
  1. Connecting to the MIC card for the first time
  2. Copying in data for an offload region
  3. Copying out data for an offload region
- Parallel regions in offload mode may run slower than in native mode

Offload Mode Issues

- Currently cannot persist automatic arrays on the stack across offload regions
  - Even though array **a** is local to subroutine **bar**, it still needs to be "copied in":

```
program foo
  call bar(size1)
end program

subroutine bar(size1)
  ! local (automatic) array a
  real, dimension(size1) :: a(:)
end subroutine bar
```



```
program foo
  ! array a now global (heap)
  real, allocatable, dimension(:) :: a
  allocate(a(size1))
  call bar(size1, a)
end program

subroutine bar(size1, a)
  ! array a turned into dummy var
  real, dimension(size1) :: a(:)
end subroutine bar
```

  - Solution: Restructure code to move local arrays inside layers of subroutines to the heap
- Intel is aware of this issue and is addressing it for the next software releases

MIC Offload vs. PGI Accelerator Model

(Similarities)

- Both approaches only require additional directives and possibly some code transformations
  - No large-scale code refactoring/rewriting

Intel MIC offload keywords	PGI ACC data region keywords
`in`	`copyin`
`out`	`copyout`
`inout`	`copy`
`nocopy`	`local`

- Both compilers report:
  - what data is being moved in and out of each offload region
  - which loops have been successfully vectorized/parallelized

MIC Offload vs. PGI Accelerator Model

(Differences)

- Persisting data across offload regions:
  - MIC: user must specify which vars need to be retained for the next offload region
  - PGI ACC: user needs to create an encompassing "data region" to persist data across individual "compute regions"
- Subroutine calls within offload regions:
  - MIC: allowed
  - PGI ACC: allowed within data regions, but not within compute regions
- Running on the co-processor:
  - MIC: offload code will still run (slowly) even if it does not vectorize/parallelize
  - PGI ACC: will refuse to generate GPU kernels unless:
    - loop carried dependencies are removed
    - certain arrays are declared `private`
    - no live variables after parallel loops
    - etc.
- Generally, since MIC card is also x86, there is less tuning than for PGI ACC running on GPUs
  - PGI ACC may require larger code transformations to expose lots of fine-grained parallelism

Outline

- What is Intel's MIC Platform?
- MIC Hardware
- MIC Programming Models
  - Offload Mode
  - Native Mode
- MIC SGEMM Performance
- FV 2D Advection Code
- Conclusion

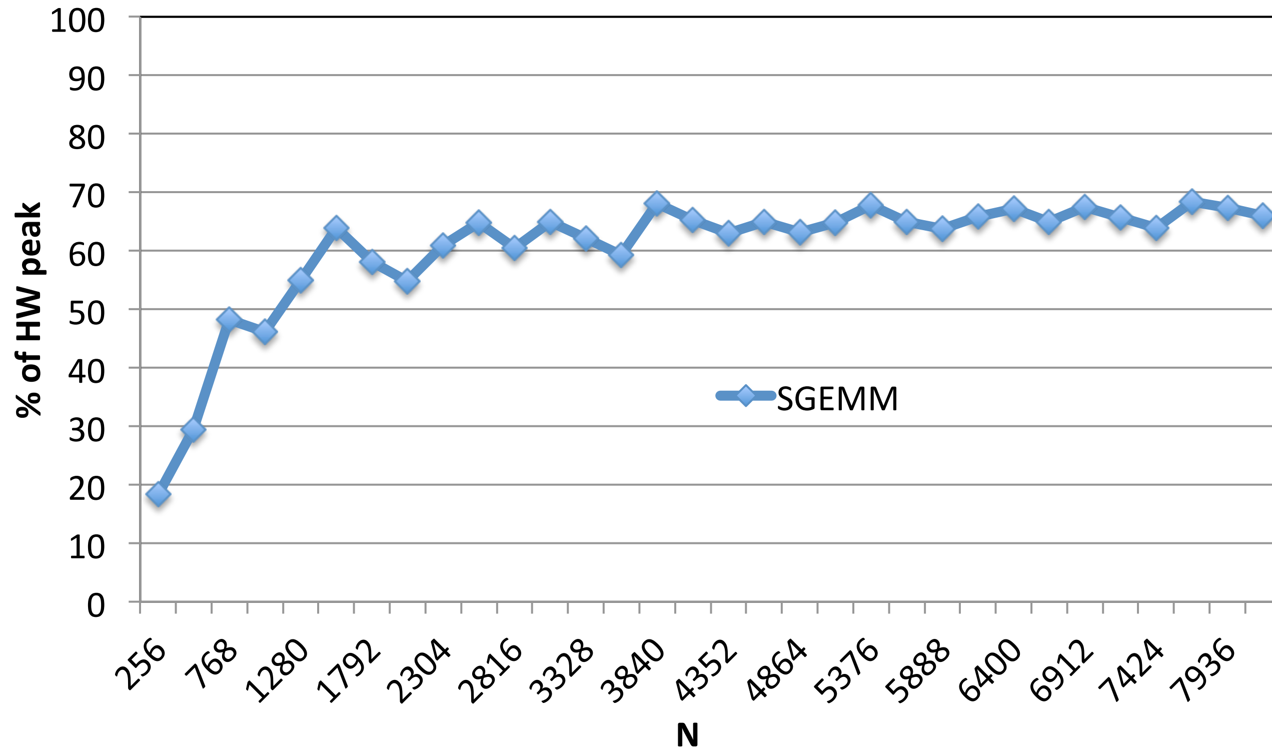
Native Mode Basics

- ***Everything runs on the MIC card***
  - no need for offload directives
  - codes with large serial regions will suffer
- OpenMP parallel regions will parallelize over MIC cores
- Code compilation:
  - ***can build as is without any code changes***
  - requires special native mode flags
    - -opt-subscript-in-range -align all **-mmic** -openmp -O3
  - generates report to show whether offloaded code is vectorized or not
- Can use OMP_NUM_THREADS, KMP_AFFINITY, and MKL libraries (just like offload mode)
- Code execution:
  1. use ssh to remotely launch a native executable on MIC card, or:
  2. ssh to MIC card, copy the executable over from host, and run natively

Outline

- What is Intel's MIC Platform?
- MIC Hardware
- MIC Programming Models
- MIC SGEMM Performance
- FV 2D Advection Code
- Conclusion

MIC SGEMM Performance



- Code was run natively on single MIC card
- Attains up to 68% of hardware peak

Outline

- What is Intel's MIC Platform?
- MIC Hardware
- MIC Programming Models
- MIC SGEMM Performance
- FV 2D Advection Code
  - Experience Porting to MIC
  - Performance
- Conclusion

OpenMP Parallelization

Original code

```
program foo
  real, dimension(:,:), allocatable a
  allocate a(size1,size2)
  do iter = 1, numTimeSteps
     call bar(a, size1, size2)
  enddo
end program foo

module bar
  contains
  subroutine bar(a, size1, size2)
     real, dimension(size1,size2) :: a
     ! local
     real, dimension(size1,size2) :: b
     some_parallel_work
     call bar1(b, size1, size2)
     more_parallel_work
     more_calls
     etc
  end subroutine bar

  subroutine bar1(b, size1, size2)
     real, dimension(size1,size2) :: b
     ! local
     real, dimension(size1,size2) :: c
     some_parallel_work
     calls_to_other_inner_subroutines
     more_parallel_work
     etc.
  end subroutine bar1
  etc.
end module bar
```

OpenMP "parallel do" directives allow do loops to be parallelized

Each "parallel do" region can also become a MIC offload region (with appropriate directives)

Original code parallelized with OpenMP

```
program foo
  real, dimension(:,:), allocatable a
  allocate a(size1,size2)
  do iter = 1, numTimeSteps
     call bar(a, size1, size2)
  enddo
end program foo

module bar
  contains
  subroutine bar(a, size1, size2)
     real, dimension(size1,size2) :: a
     ! local
     real, dimension(size1,size2) :: b
     !$omp parallel do
     some_parallel_work
     call bar1(b, size1, size2)
     !$omp parallel do
     more_parallel_work
     more_calls
     etc
  end subroutine bar

  subroutine bar1(b, size1, size2)
     real, dimension(size1,size2) :: b
     ! local
     real, dimension(size1,size2) :: c
     !$omp parallel do
     some_parallel_work
     calls_to_other_inner_subroutines
     !$omp parallel do
     more_parallel_work
     etc.
  end subroutine bar1
  etc.
end module bar
```

Advection Code on the MIC

(Offload Mode)

Original code parallelized with OpenMP

```
program foo
  real, dimension(:,:), allocatable a
  allocate a(size1,size2)
  do iter = 1, numTimeSteps
     call bar(a, size1, size2)
  enddo
end program foo

module bar
  contains
  subroutine bar(a, size1, size2)
     real, dimension(size1,size2) :: a
     ! local
     real, dimension(size1,size2) :: b
     !$omp parallel do
     some_parallel_work
     call bar1(b, size1, size2)
     !$omp parallel do
     more_parallel_work
     more_calls
     etc
  end subroutine bar

  subroutine bar1(b, size1, size2)
     real, dimension(size1,size2) :: b
     ! local
     real, dimension(size1,size2) :: c
     !$omp parallel do
     some_parallel_work
     calls_to_other_inner_subroutines
     !$omp parallel do
     more_parallel_work
     etc.
  end subroutine bar1
  etc.
end module bar
```

We could offload each "parallel do" region to the MIC, but local arrays would not persist

As a result, we move local arrays to the top-most caller subroutine, and then create a single parallel offload region

MIC code

```
program foo
  real, dimension(:,:), allocatable a, b, c
  !dir$ attributes offload : mic :: a, b, c
  allocate a(size1,size2)
  allocate b(size1,size2)
  allocate c(size1,size2)

  !dir$ offload target(mic:0) in(b, c) inout(a)
  !$omp parallel
  do iter = 1, numTimeSteps
     call bar(a, b, c, size1, size2)
  enddo
  !$omp end parallel
end program foo

module bar
  contains
  subroutine bar(a, b, c, size1, size2)
     real, dimension(size1,size2) :: a, b, c
     !$omp do
     some_parallel_work
     call bar1(b, c, size1, size2)
     !$omp do
     more_parallel_work
     more_calls
     etc
  end subroutine bar

  subroutine bar1(b, c, size1, size2)
     real, dimension(size1,size2) :: b, c
     !$omp do
     some_parallel_work
     calls_to_other_inner_subroutines
     !$omp do
     more_parallel_work
     etc.
  end subroutine bar1
  etc.
end module bar
```

Advection Code on the MIC

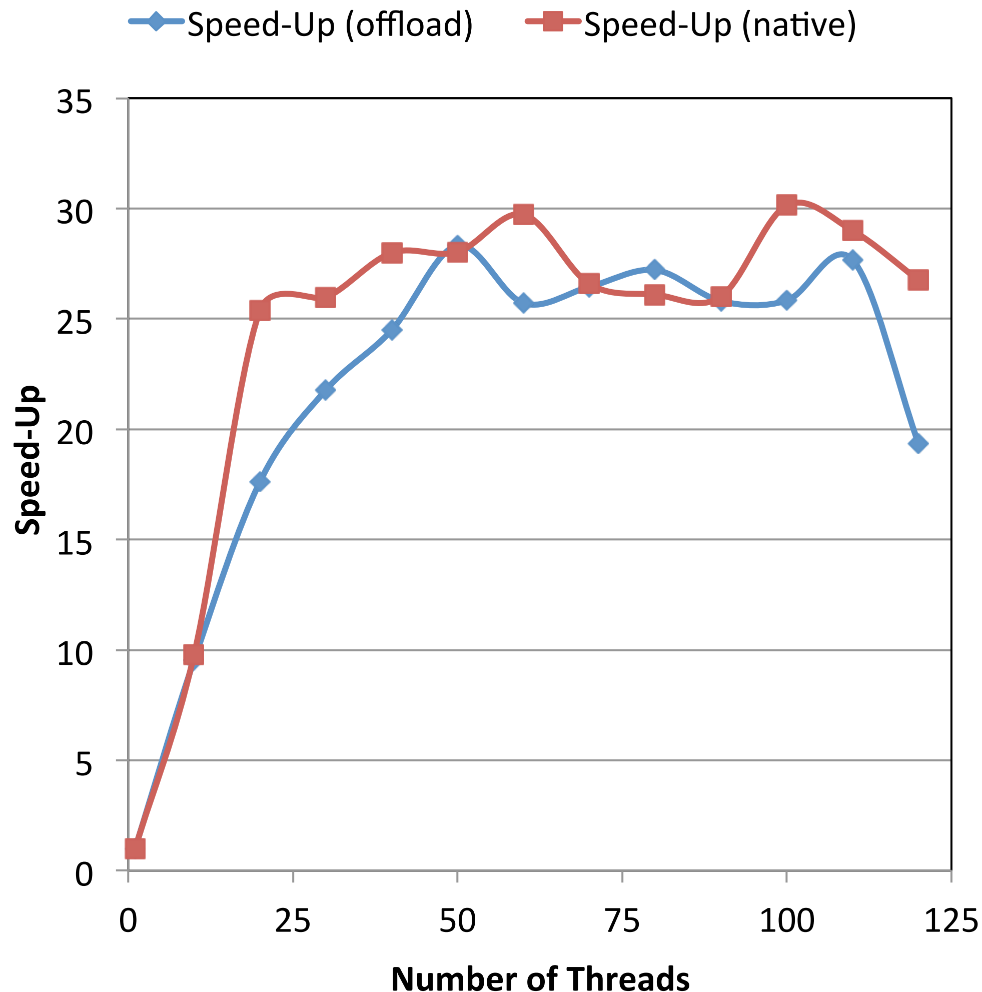
(Steps Taken for Offload Mode)

- Code reorganized to put automatic variables (**`b`**, **`c`**) on the heap and pass them as arguments
- Code restructured to have only single "**`omp parallel`**" at highest level
  - Allows MIC data to be copied only once
- MIC directives then added above "**`omp parallel`**" directive to offload work and data
- For OpenMP performance:
  - Environmental variable KMP_AFFINITY should be set to "granularity=thread,scatter"

Outline

- What is Intel's MIC Platform?
- MIC Hardware
- MIC Programming Models
- MIC SGEMM Performance
- FV 2D Advection Code
  - Experience Porting to MIC
  - Performance
- Conclusion

2D Advection Code Performance



- Speed-up basically flattens out at a maximum of:
  - about 28 at 50 threads for "offload"
  - about 30 at 60 (and 100) threads for "native"
- Not shown but observed:
  - "native" about 30% faster than "offload"
  - Cost of spawning threads grows with number of threads in "native" mode but remains constant in "offload" mode

Outline

- What is Intel's MIC Platform?
- MIC Hardware
- MIC Programming Models
- MIC SGEMM Performance
- FV 2D Advection Code
- Conclusion

Conclusions

- Once automatic variables persist on the MIC, we expect porting of parallel codes to be quicker than for GPUs (typically within days)
  - Similar programming and environment between CPUs and MIC
  - “Offload” mode might only require addition of offload directives
  - “Native” mode might not require any code changes
  - Maintain a single code base?
  - Generating GPU kernels using PGI ACC still not easy
- Have yet to see how Knights Corner performance compares to latest GPUs