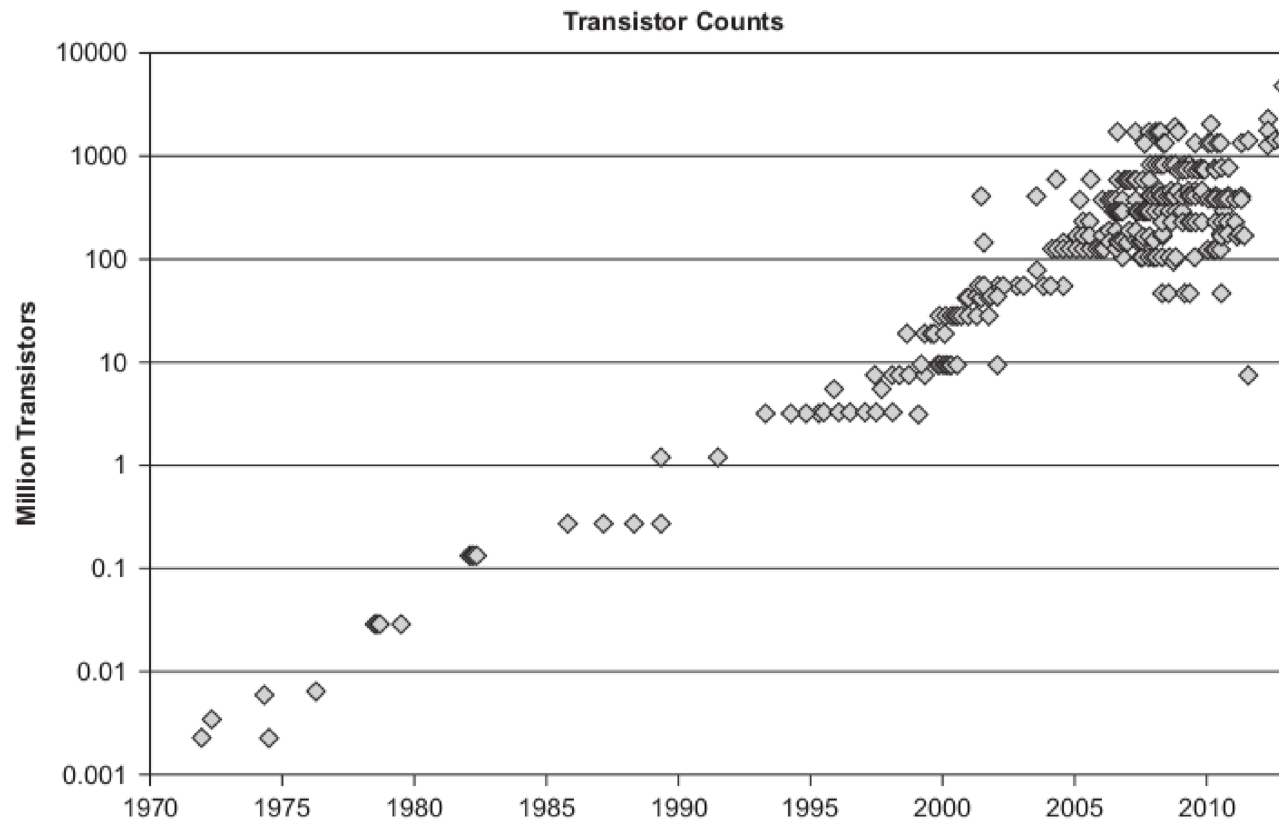
Introduction to Intel Xeon Phi programming models

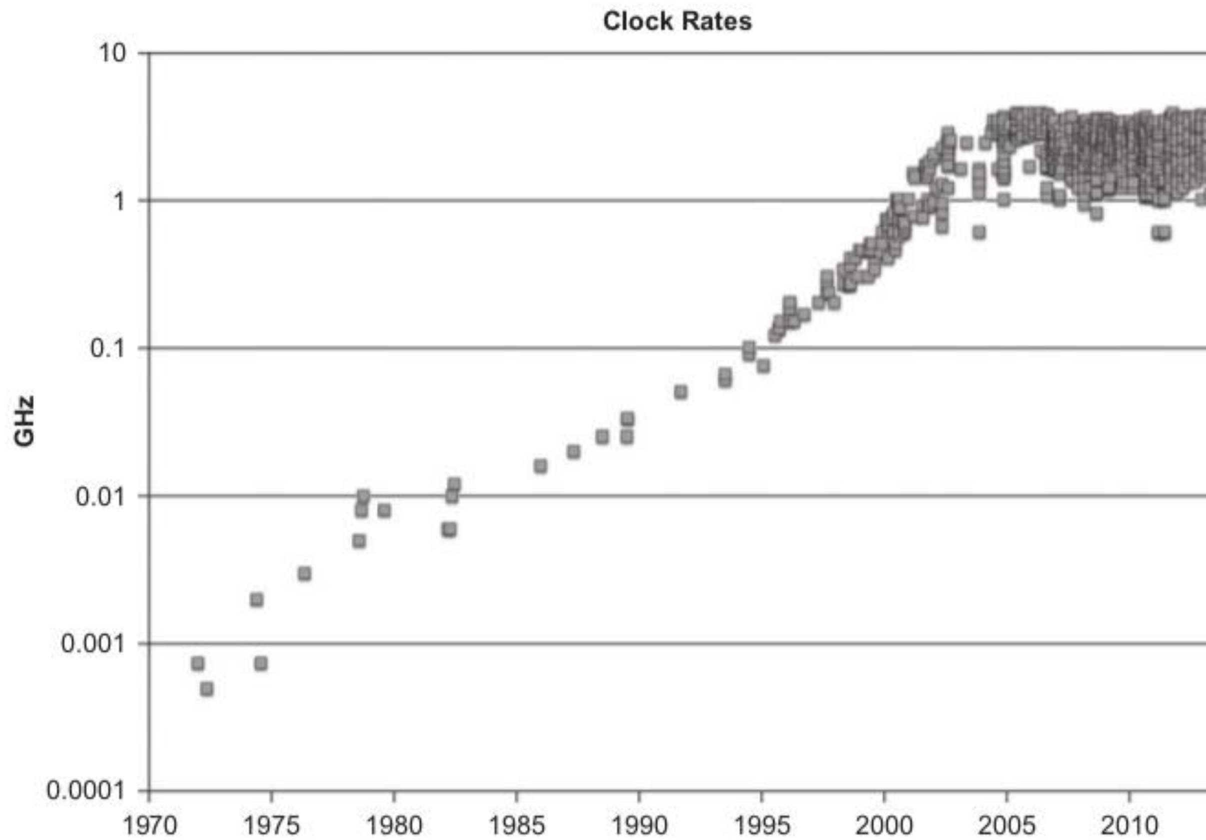
F.Affinito
F. Salvadore
SCAI - CINECA

Part I
Introduction to the Intel Xeon Phi architecture

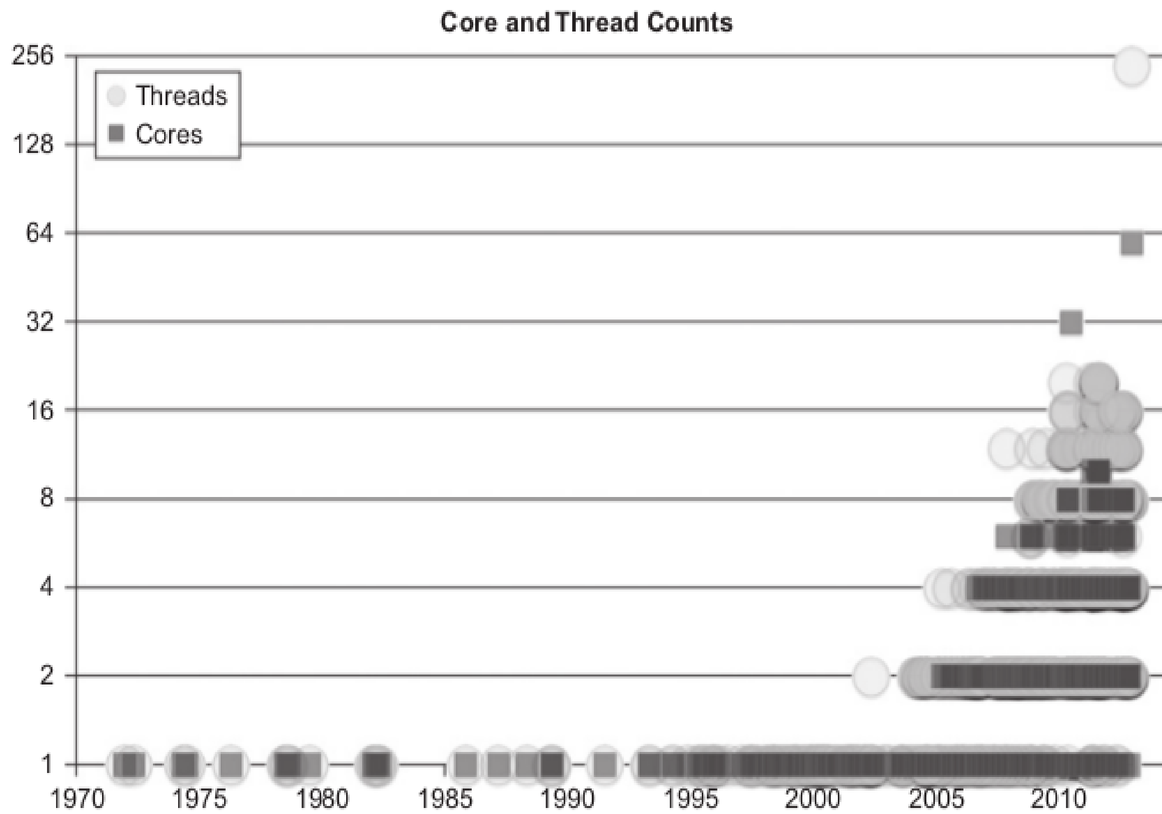
Trends: transistors



Trends: clock rates



Trends: cores and threads



Trends: summarizing...

- The number of transistors increases
- The power consumption must not increase
- The density cannot increase on a single chip

Solution :

- Increase the number of cores

GP-GPU and Intel Xeon Phi..

- Coupled to the CPU
- To accelerate highly parallel kernels, facing with the Amdahl Law

What is Intel Xeon Phi?

- 7100 / 5100 / 3100 Series available
- 5110P:
  - Intel Xeon Phi clock: 1053 MHz
  - 60 cores in-order
  - ~ 1 TFlops/s DP peak performance (2 Tflops SP)
  - 4 hardware threads per core
  - 8 GB DDR5 memory
  - 512-bit SIMD vectors (32 registers)
  - Fully-coherent L1 and L2 caches
  - PCIe bus (rev. 2.0)
  - Max Memory bandwidth (theoretical) 320 GB/s
  - Max TDP: 225 W

MIC vs GPU *naïve* comparison

- The comparison is naïve

System	K20s	5110P
# cores	2496	60 (*4)
Memory size	5 GB	8 GB
Peak performance (SP)	3.52 TFlops	2 TFlops
Peak performance (DP)	1.17 TFlops	1 TFlops
Clock rate	0.706 GHz	1.053 GHz
Memory bandwidth	208 GB/s (ECC off)	320 GB/s

Terminology

- **MIC** = Many Integrated Cores is the name of the architecture
- **Xeon Phi** = Commercial name of the Intel product based on the MIC architecture
- **Knight's corner,** Knight's landing, Knight's ferry are development names of MIC architectures
- We will often refer to the CPU as **HOST** and Xeon Phi as **DEVICE**

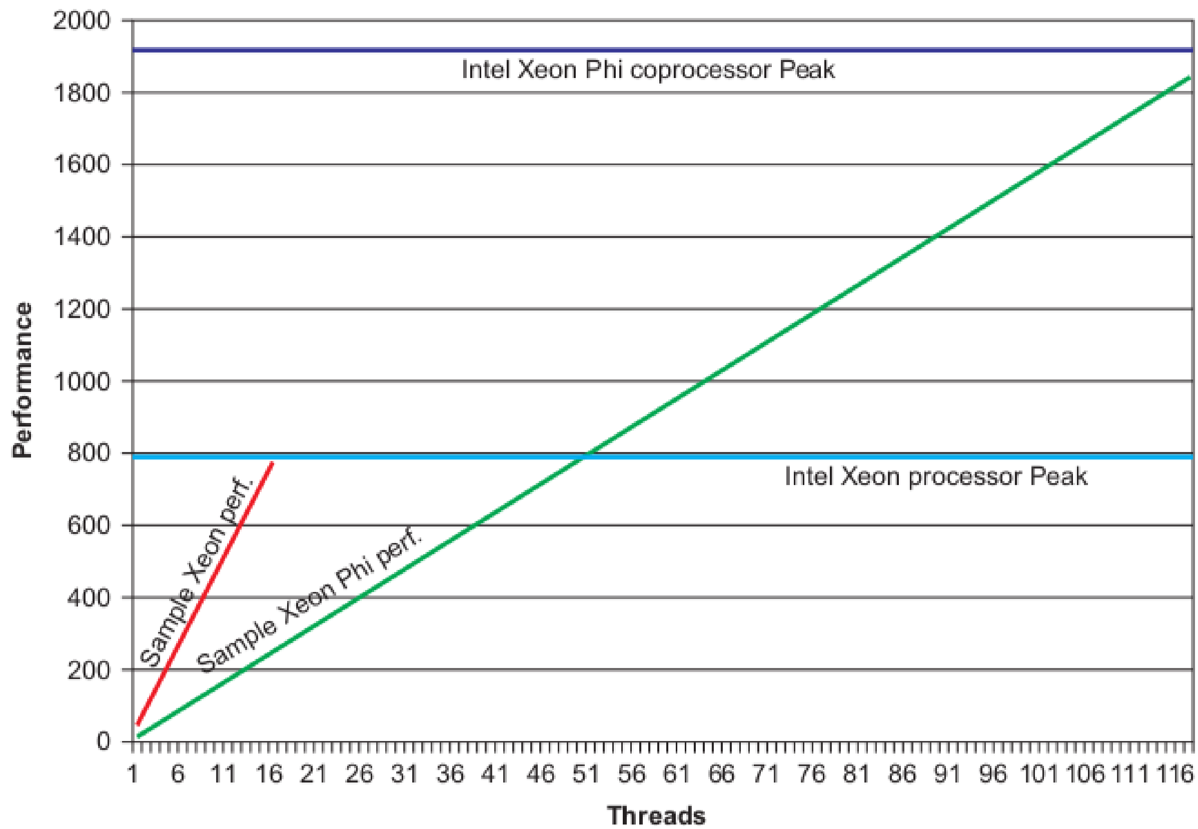
Is it an accelerator?

- YES: It can be used to “accelerate” hot-spots of the code that are highly parallel and computationally extensive
- In this sense, it works alongside the CPU
- It can be used as an accelerator using the “offload” programming model
- An important bottleneck is represented by the communication between host and device (through PCIe)
- Under this respect, it is very similar to a GPU

Is it an accelerator? / 2

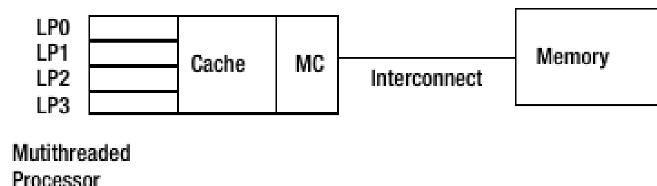
- NOT ONLY: the Intel Xeon Phi can behave as a many-core X86 node.
  - Code can be compiled and run “natively” on the Xeon Phi platform using MPI + OpenMP
- The bottleneck is the scalability of the code
  - Amdahl Law
- Under this respect, the Xeon Phi is completely different from a GPU
  - This is way we often call the Xeon Phi “co-processor” rather than “accelerator”

Many-core performances



Architecture key points/1

- Instruction Pipelining
  - Two independent pipelines arbitrarily known as the U and V pipelines
  - (only) 5 stages to cope with a reduced clock rate, e.g. compared to the Pentium 20 stages
  - In-order instruction execution
- Manycore architecture
  - Homogeneous
  - 4 hardware threads per core



Architecture key points/2

- Interconnect: bidirectional ring topology
  - All the cores talk to one another through a bidirectional interconnect
  - The cores also access the data and code residing in the main memory through the ring connecting the cores to memory controller
- Given eight memory controllers with two GDDR5 channels running at 5.5 GT/s
  - Aggregate Memory Bandwidth = 8 memory controllers × 2 channels × 5.5 GT/s × 4 bytes/transfer = 352 GB/s
- System interconnect
  - Xeon Phi are often placed on PCIe slots to work with the host processors

Architecture key points/3

- Cache:
  - L1: 8-ways set-associative 32-kB instruction and 32-kB data
  - L1 access time: 3 cycles
  - L2: 8-way set associative and 512 kB in size (unified)Interconnect: bidirectional ring topology
- TLB cache:
  - L1 data TLB supports three page sizes: 4 kB, 64 kB, and 2 MB
  - L2 TLB
  - If one misses L1 and also misses L2 TLB, one has to walk four levels of page table, which is pretty expensive

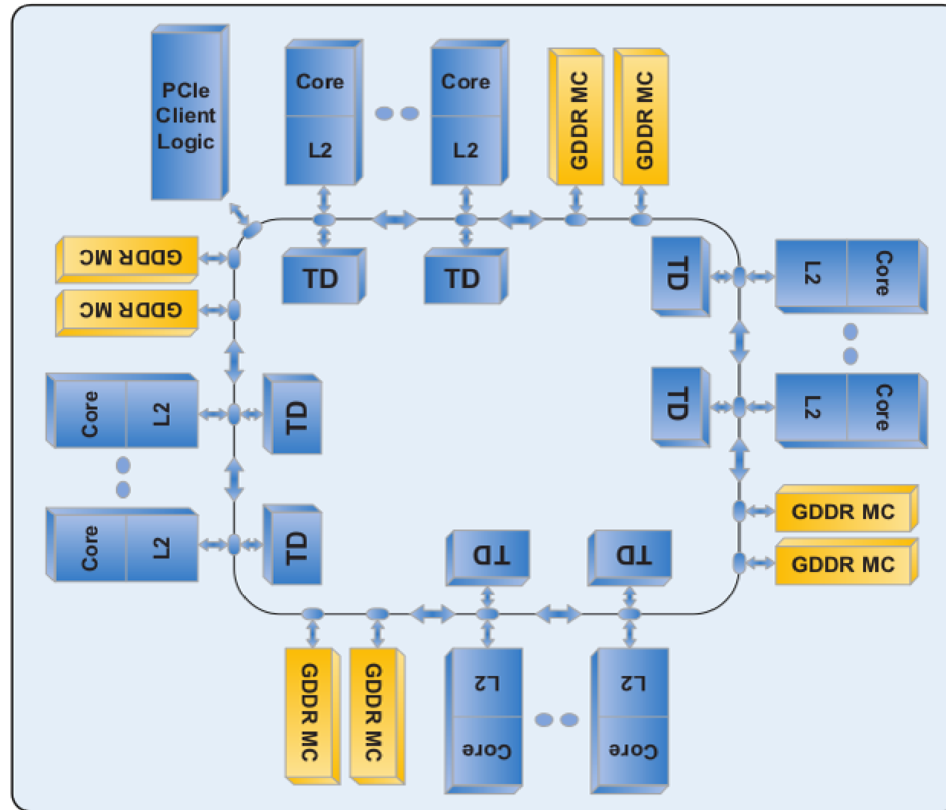
Architecture key points/4

- The VPU (vector processing unit) implements a novel instruction set architecture (ISA), with 218 new instructions compared with those implemented in the Xeon family of SIMD instruction sets.
- The VPU is fully pipelined and can execute most instructions with four-cycle latency and single-cycle throughput.
- Each vector can contain 16 single-precision floats or 32-bit integer elements or eight 64-bit integer or double-precision floating point elements.

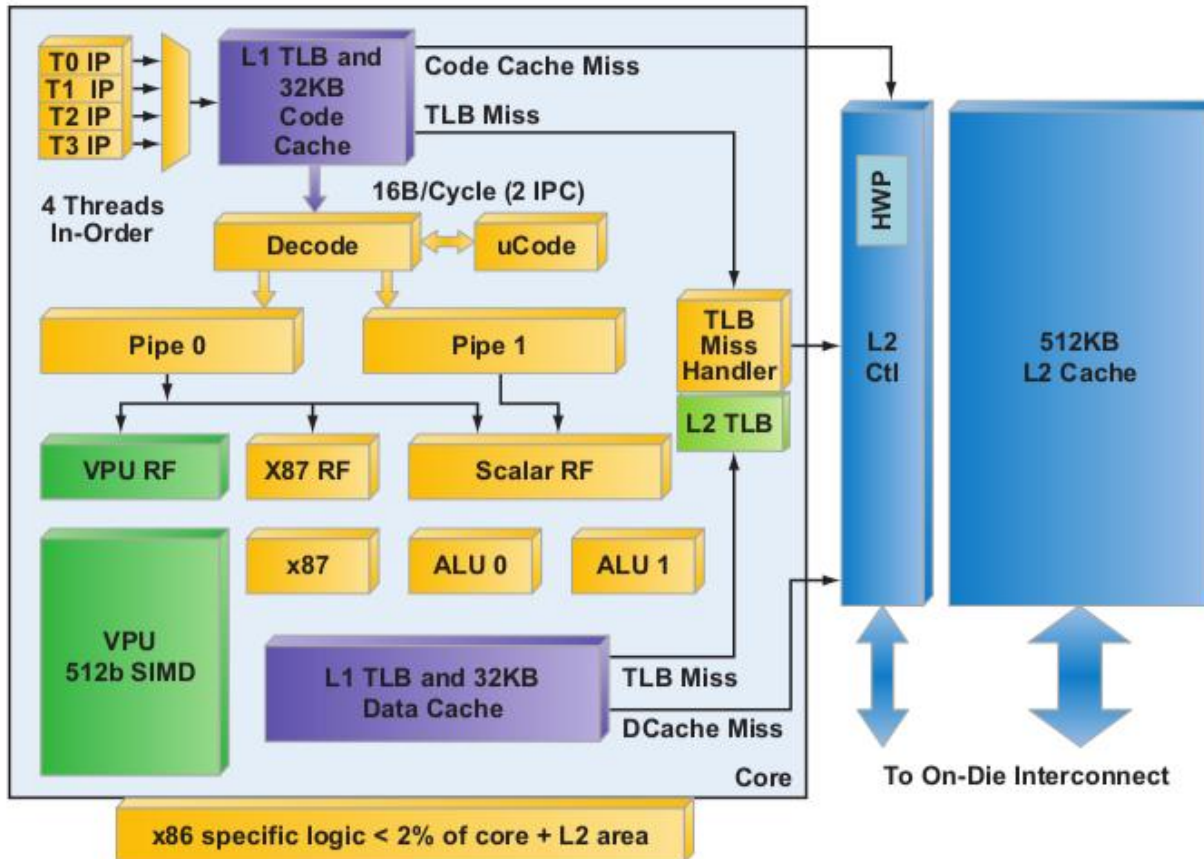
Architecture key points/5

- Each VPU instruction passes through one or more of the following five pipelines to completion:
  - Double-precision (DP) pipeline: Used to execute float64 arithmetic, conversion from float64 to float32, and DP-compare instructions.
  - Single-precision (SP) pipeline: Executes most of the instructions including 64-bit integer loads. This includes float32/int32 arithmetic and logical operations, shuffle/broadcast, loads including loadunpack, type conversions from float32/int32 pipelines, extended math unit (EMU) transcendental instructions, int64 loads, int64/float64 logical, and other instructions.
  - Mask pipeline: Executes mask instructions with one-cycle latencies.
  - Store pipeline: Executes the vector store operations.
  - Scatter/gather pipeline: Executes the vector register read/writes from sparse memory locations.
- Mixing SP and DP computations is expensive!

Architecture sketch/1



Architecture sketch/2



Part II
Programming models

Same target, different pathways

Though the problem is very similar (i.e. heterogeneous device connected to the CPU through a low bandwith channel), there are many programming approaches:

-Based on a low-level language addressing the memory space of the device
- -Proprietary (CUDA)
- -Open (OpenCL)

-Based on directives (“pragma”) to the compiler
- -OpenMP4.0
- -OpenACC
- -Intel Language Extension for Offload (LEO)

Comparing strategies

- Directive based approach are relatively simpler because they permit to manage the data transfer in an explicit way without affecting too much the original code
- Directive based approaches preserve the portability of the code

- CUDA and OpenCL are low level languages that permits to obtain better perfomances but:
    - CUDA is a proprietary language and it is only suitable for GPUs
    - OpenCL is open and suitable for both GPUs and MICs but it does not permit to reach the same level of performances for both of them
    - Both CUDA and OpenCL require to make important changes to the code
- Using CUDA in many cases forces to create a separate developing branch of the code.

- Intel LEO can be used only on the Intel Xeon Phi platform
    - Intel LEO approach is much similar to the directives implemented in OpenMP4.0
    - Implementing LEO forces to better work on vectorization and threadization: this effort is fruitful also when working on CPU only

Intel Xeon Phi programming models

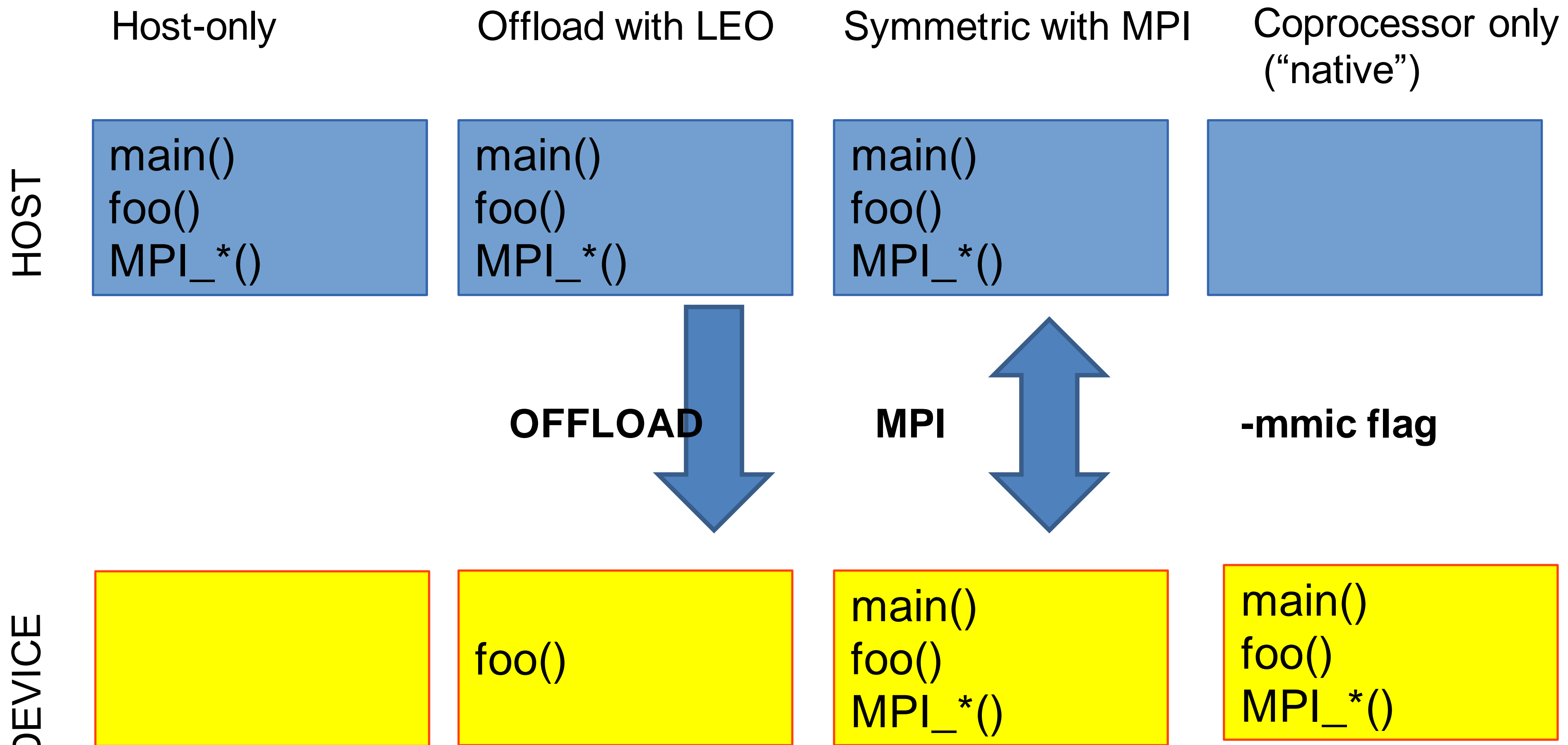
Intel Xeon Phi has a twofold nature:

- accelerator
- coprocessor

According to this one has the chance to work with different programming models.

The choice of the programming model can be made according to the requirements of the application.

Intel Xeon Phi programming models



Intel Xeon Phi programming models

PRO:

- it is a cross-compiling mode
- very simple
  - just add -mmic, login and execute
- use well known OpenMP and MPI (or pthreads or OpenCL)

CONS:

- very slow I/O
- poor single thread performance
- only suitable for highly parallel codes (cfr Amdahl)
- CPU unused

Intel Xeon Phi programming models

Intel provides a set of directives to the compiler named “LEO”: Language Extension for Offload.

These directives manage the transfer and execution of portions of code to the device.

C/C++

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

Fortran

```
!dir$ offload target (mic:device_id)
```

Intel Xeon Phi programming models

Variable and function definitions

C/C++
__attribute__ ((target(mic)))

Fortran
!dir$ attributes offload:mic :: <function/var name>

It compiles (allocates) variables on both the host and device

For entire files or large blocks of code (C/C++ only)
#pragma offload_attribute (push, target(mic))
#pragma offload_attribute (pop)

Intel Xeon Phi programming models

Since host and device don't have physical or virtual shared memory, variable must be copied in an explicit or in an implicit way.

Implicit copy is assumed for
- scalar variables
- static arrays

Explicit copy must be managed by the programmer using clauses defined in the LEO

Intel Xeon Phi programming models

Programmer clauses for explicit copy:
in, out, inout, nocopy

Data transfer with offload region:

```
C/C++     #pragma offload target(mic) in(data:length(size))
Fortran   !dir$ offload target (mic) in(data:length(size))
```

Data transfer without offload region:

```
C/C++ #pragma offload_transfer target(mic)in(data:length(size))
Fortran  !dir$ offload_transfer target(mic) in(data:length(size))
```

Intel Xeon Phi programming models: examples

C/C++

```
#pragma offload target (mic) out(a:length(n)) \
in(b:length(n))
for (i=0; i<n; i++){
        a[i] = b[i]+c*d
}
```

Fortran

```
!dir$ offload begin target(mic) out(a) in(b)
do i=1,n
        a(i)=b(i)+c*d
end do
!dir$ end offload
```

Intel Xeon Phi programming models: examples

C/C++

```
__attribute__ ((target(mic)))
void foo(){
        printf("Hello MIC\n");
}

int main(){
#pragma offload target(mic)
        foo();
return 0;
}
```

Fortran

```
!dir$  attributes &
!dir$  offload:mic ::hello
subroutine hello
        write(*,*)"Hello MIC"
end subroutine

program main
!dir$ attributes &
!dir$ offload:mic :: hello
!dir$ offload begin target (mic)
        call hello()
!dir$ end offload
end program
```

Intel Xeon Phi programming models

Memory allocation

- CPU is managed as usual
- on coprocessor is defined by in,out and inout clauses

Input/Output pointers

- by default on coprocessor "new" allocation is performed for each pointer
- by default de-allocation is performed after offload region
- defaults can be modified with alloc_if and free_if qualifiers

Intel Xeon Phi programming models

Using memory qualifiers

free_if(0)
free_if(.false.) retain target memory

alloc_if(0)
alloc_if(.false.) reuse data in subsequent offload

alloc_if(1)
alloc_if(.true.) allocate new memory

free_if(1)
free_if(.true.) deallocate memory

Intel Xeon Phi programming models

note: specify the device_id when using more than one device
target(mic:0)
target(mic:1) ...

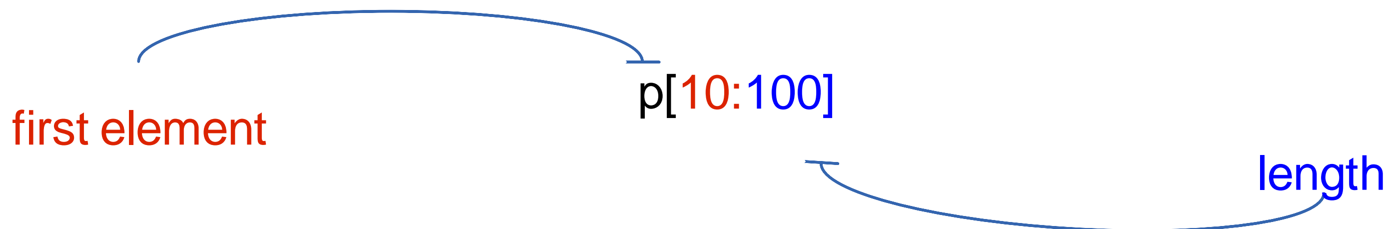
```
#define ALLOC alloc_if(1)
#define FREE free_if(1)
#define RETAIN free_if(0)
#define REUSE alloc_if(0)

#allocate the memory but don't de-allocate
#pragma offload target(mic:0) in(a:length(8)) ALLOC RETAIN)
...

#don't allocate or deallocate the memory
#pragma offload target(mic:0) in(a:length(8) REUSE RETAIN)

#don't allocate the memory but de-allocate
#pragma offload target(mic:0) in(a:length(8) REUSE FREE)
```

Intel Xeon Phi programming models

Partial offload of arrays

```
int *p;
#pragma offload ... in (p[10:100] : alloc(p(5:1000))
{...}
```

It allocates 1000 elements on coprocessor; first usable element has index 5, last has index 1004; only 100 elements are tranferred,
starting from index 10.



Intel Xeon Phi programming models

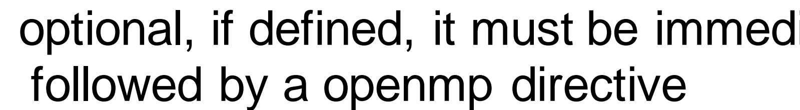
Copy from a variable to another one

It permits to copy data from the host to a different array allocated on the device

```
integer :: p(1000), p1(2000)
integer :: rank1(1000), rank2(10,100)

!dir$ offload ... (p(1:500) : into (p1(501:1000)))
```

Intel Xeon Phi programming models

Using OpenMP in an offload region:




```
C/C++
#pragma offload target (mic)
#pragma omp parallel for
for (i=0; i<n; i++){
        a[i]=b[i]*c+d;
}
```

```
Fortran
!dir$ omp offload target (mic)
!$omp parallel do
do i=1,n
        A(i)=B(i)*C+D
end do
!$omp end parallel
```

Setting up the environment:
OMP_NUM_THREAD = 16
MIC_ENV_PREFIX = MIC
MIC_OMP_NUM_THREADS = 120

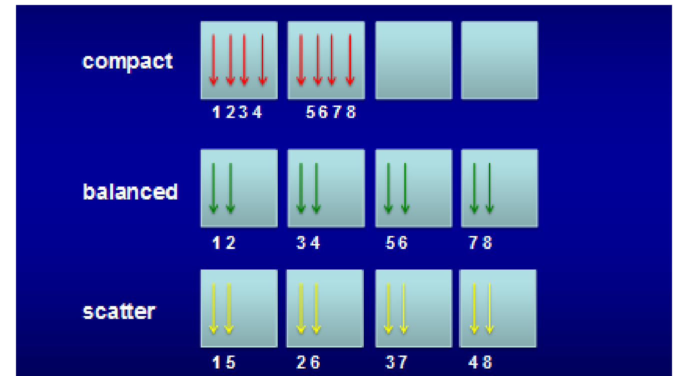
Intel Xeon Phi programming models

Tuning up OpenMP

the coprocessor has 4 hardware threads per core; at least 2 should be used;
for many-cores systems (and hence for the Xeon Phi) binding threads to the cores and choosing an affinity are crucial factor that affect performance

MIC_ENV_PREFIX = MIC
MIC_KMP_AFFINITY =

- compact
- scattered
- balanced



Intel Xeon Phi programming models

LEO compiler options

- -opt-report-phase:offload activate reporting
- -no-offload disable offload
- -offload-attribute-target=mic build ALL functions for both host and device

Intel Xeon Phi programming models

Static libraries

- xiar can be used to create libraries containing offloaded code
- specify -qoffload-build that forces xiar to create both a library for the host (xxx.a) and a library for the device (xxxMIC.a)
- use the same options that you would use for ar
- use normally the linker options (-L.. -lxxx.a) and the compiler will automatically include the coprocessor library

Intel Xeon Phi programming models

Managing multiple devices

Including offload.h

```
#include <offload.h>
```

you can use a few API:

```
_offload_number_of_device()
_offload_get_device_number()
```

or use runtime environmente variables:

```
OFFLOAD_DEVICES=0,1
```

always remember to specify the target device target(mic:1)

Intel Xeon Phi programming models

MIC specific MACROs

```
#ifdef __INTEL_OFFLOAD
#include <offload.h>
#endif

#ifdef __INTEL_OFFLOAD
  printf("%d MICS available\n",_Offload_number_of_devices());
#endif

int main(){
#pragma offload target(mic)
  {
#ifdef __MIC__
  printf("Hello MIC number %d\n", _Offload_get_device_number());
#else
  printf("Hello HOST\n");
  }
}
```

Intel Xeon Phi programming models

I/O from offloaded regions

- Buffered printf are available (use only to debugging purposes!)
- Always use fflush(0) on the coprocessor

- Files I/O is possible only through a proxy filesystem

Intel Xeon Phi programming models

Asynchronous computation

By default, offload forces the host to wait for completion

- Asynchronous offload starts the offload and continues on the next statement just after the offload region
- Use the signal clause to synchronize with a offload_wait statement

Intel Xeon Phi programming models

Example

```
char signal_var;
do {
  #pragma offload target(mic:0) signal(&signal_var)
  {
     long_running_mic_compute();
  }
  concurrent_cpu_computation();
  #pragma offload_wait target(mic:0) wait(&signal_var)
} while(1);
```

Intel Xeon Phi programming models

Reporting

Use OFFLOAD_REPORT or the variable _Offload_report with a verbosity from 1 to 3.
OFFLOAD_REPORT=1 only provides timing

Conditional offload

Only offload if it is worth

```
#pragma offload target (mic) in (b:length(size)) \
                    out (a:length(size) \
                    if(size>100)
```

Intel Xeon Phi programming models: native mode

```
icc -mmic mycode.c -o mycode.x
scp mycode.x mic0:.
ssh mic0
export OMP_NUM_THREADS=240
./mycode.x
```

Simple... but not always successful

Intel Xeon Phi programming models: symmetric mode

Using MPI you can make work together the executable running on the host and the one running on the device (compiled with -mmic)

- Load balancing can be an issue
- Tuning of MPI and OpenMP on both host and device is crucial
- Dependent on the cluster implementation (physical network, MPI implementation, job scheduler..)

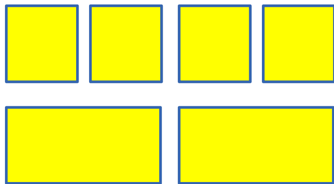
Intel Xeon Phi programming models: symmetric mode

Example

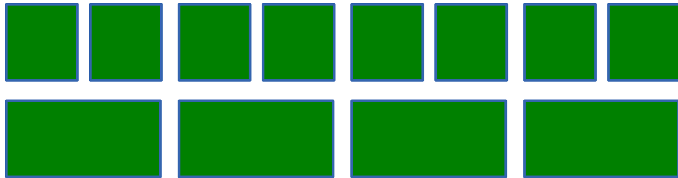
```
# compile the program for the coprocessor (-mmic)
mpiicc -mmic -o test.MIC test.c
# compile the program for the host
mpiicc -mmic -o test test.c

#copy the executable to the coprocessor
scp test.MIC mic0:/tmp/test.MIC

#set the I_MPI_MIC variable
export I_MPI_MIC=1

#launch MPI jobs on the host knf1 and on the coprocessor mic0
mpirun -host knf1 -n 1 ./test : -n 1  -host mic0 /tmp/test.MIC
```

Vectorization

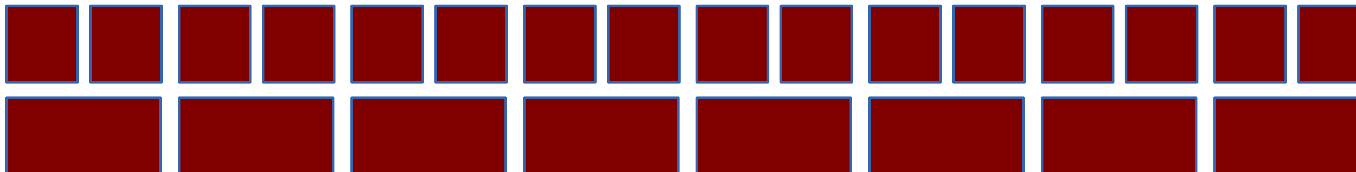
Lots of cores but also... large registers!



SSE : 128 bit
2 x DP or 4 x SP



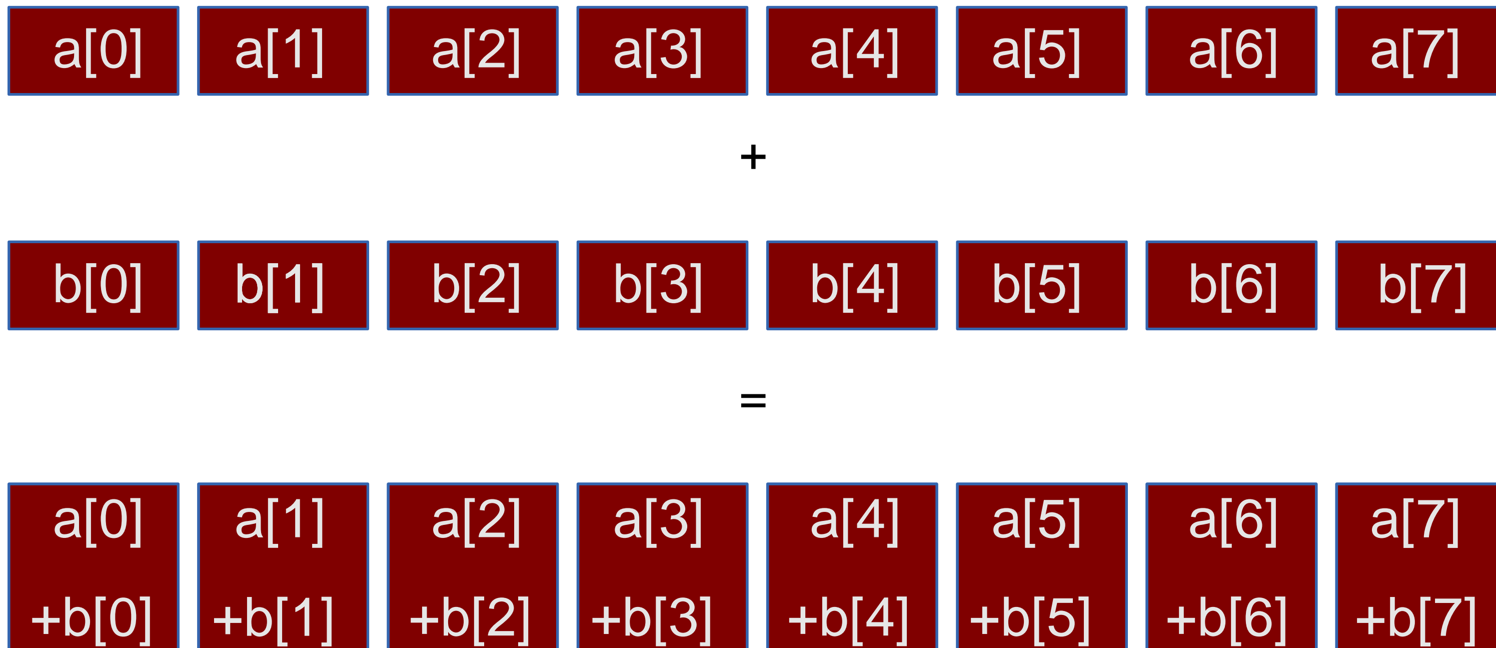
AVX : 256 bit
4 x DP or 8 x SP



MIC : 512 bit
8 x DP or 16 x SP

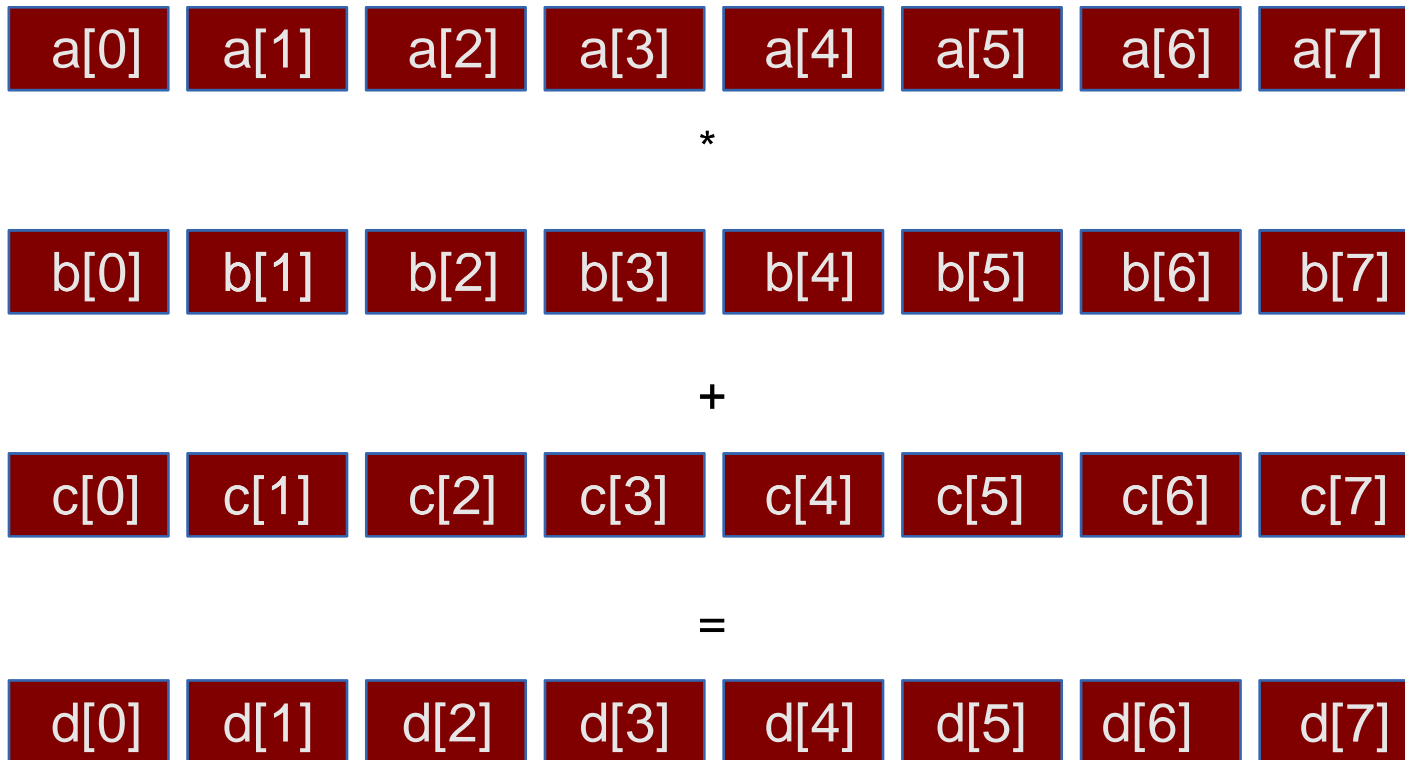
Vectorization

SIMD arithmetic



Vectorization

SIMD Fused Multiply Add



MKL Libraries

Intel released a version for Xeon Phi of the MKL mathematical libraries

MKL have three different usage models

- Automatic offload (AO)
- Compiler assisted offload (CAO)
- Native execution

MKL Libraries

- Offload is automatic and transparent
- The library decides **when** to offload and **how much** to offload (workdivision)
- Users can control parameters through environment variables or API

You can enable automatic offload with
MKL_MIC_ENABLE=1
or
mkl_mic_enable()

MKL Libraries

Not all the MKL functions are enabled to AO.

In MKL 11.0.1:

- *Level 3 BLAS*: xGEMM, xTRSM, xTRMM
- *LAPACK* xGETRF, xPOTRF, xGEQRF

Always check the documentation for updates

MKL Libraries

- MKL functions can be offloaded as other “ordinary” functions using the LEO pragmas
- All MKL functions can take advantage of the CAO
- It's a more flexible option in terms of data management (you can use data persistence or mechanisms to hide the latency...)

MKL Libraries: CAO

C/C++

```
#pragma offload target (mic) \
in (transa, transb, N, alpha, beta) \
in (A:length(matrix_elements)) in (B:length(matrix_elements)) \
inout (C:length(matrix_elements))
{
sgemm(&transa, &transb, &N, &N, &N, &alpha, A, &N, B, &N, &beta, C, &N);
}
```

Fortran

```
!dir$ attributes offload : mic : sgemm
!dir$ offload target(mic) &
!dir$ in (transa, transb, m, n, k, alpha, beta, lda, ldb, ldc), &
!dir$ in (a:length(ncola*lda)), in (b:length(ncolb*ldb)) &
!dir$ inout (c:length(n*ldc))
CALL sgemm (transa, transb,m,n,k,alpha,a,lda,b,ldb,beta,c,ldc)
```

MKL Libraries

MKL libraries are also available when using the native mode.

Tips:

Use all the 240 threads: MIC_OMP_NUM_THREADS=240

Set the thread affinity: MIC_KMP_AFFINITY = ...

Part III
Optimization hints

Performance and parallelism

- In principle the main advantage of using Intel MIC technology with respect to other coprocessors is the simplicity of the porting
    - Programmers may compile their source codes based on common HPC languages (Fortran/ C / C++) specifying MIC as the target architecture (*native mode*)
- Is it enough to achieve good performances? By the way, why offload?
- Usually not, parallel programming is not easy
    - A general need is to **expose parallelism**

GPU vs MIC

- GPU paradigms (e.g. CUDA):
  - Despite the sometimes significant effort required to port the codes...
  - ...are designed to force the programmer to expose (or even create if needed) parallelism
- Programming Intel MIC
  - The optimization techniques are not far from those devised for the common CPUs
  - As in that case, achieving optimal performance is far from being straightforward
- What about device maturity?

Intel Xeon Phi very basic features

- Let us recall 3 basic features of current Intel Xeon Phi:
- Peak performance originates from "many slow but vectorizable cores"

```
clock frequency x n. cores x n. lanes x 2 FMA Flops/cycle
1.091 GHz x 61 cores x 16 lanes x 2 = 2129.6 Gflops/cycle
1.091 GHz x 61 cores x 8 lanes x 2 = 1064.8 Gflops/cycle
```

- Bandwidth is (of course) limited, caches and alignment matter
- The card is not a replacement for the host processor. It is a coprocessor providing optimal power efficiency

Optimization key points

In general terms, an application must fulfill three requirements to efficiently run on a MIC

- (1) Highly vectorizable, the cores must be able to exploit the vector units. The penalty when the code cannot be vectorized is very high
- (2) high scalability, to exploit all MIC multi-threaded cores: scalability up to 240 processors (processes/threads) running on a single MIC, and even higher running on multiple MIC
- (3) ability of hiding I/O communications with the host processors and, in general, with other hosts or coprocessors

Auto-vectorization

- In recent Intel compilers, vectorization is enabled by default
  - May be turned off by explicit options
  - The compiler must be able to detect the possibility to do that
- The essential requirement is the possibility to unroll the loop having the different iterations performed simultaneously
- Some critical conditions
  - If the loop is part of a loop nest, it must be the inner loop unless it is completely unrolled or interchange occurs (use -O3)
  - Straight-line code: no jumps or branches but masked assignment allowed
  - Countable loop: number of iterations must be known when starting (even if not at compile time)
  - No loop dependencies: iterations must be performed in parallel

Vectorization: arrays and restrict

- Writing "clean" code is a good starting point to have the code vectorized
  - Prefer array indexing instead of explicit pointer arithmetic
  - Use *restrict* keyword to tell the compiler that there is no array aliasing

- Excerpt from a real code the compiler manages to vectorize:

```
REAL * __restrict__ anspx=an+spxoff;
REAL * __restrict__ ansmx=an+smxoff;
...
for(ix=istart; ix<iend; ix++) {
    as = anspx[ix]*JpxWO[ix] + anspy[ix]*JpyWO[ix] +
         anspz[ix]*JpzWO[ix] + ansmx[ix]*JmxWO[ix] +
         ansmy[ix]*JmyWO[ix] + ansmz[ix]*JmzWO[ix] +
    ...
}
```

Vectorization: array notation

- Using array notation is a good way to guarantee the compiler that the iterations are independent
  - In Fortran this is consistent with the language array syntax

```
a(1:N) = b(1:N) + c(1:N)
```

  - In C the array notation is provided by Intel Cilk Plus

```
a[1:N] = b[1:N] + c[1:N]
```

- Beware:
  - The first value represents the lower bound for both languages
  - But the second value is the upper bound in Fortran whereas it is the length in C
  - An optional third value is the stride both in Fortran and in C
  - Multidimensional arrays supported, too

Vectorization: directives

- Another opportunity is forcing vectorization by means of directives
  - The programmer guarantees the possibility to vectorize
  - Until a few years ago, only compiler dependent directives available

`#pragma ivdep`

- Instructs the compiler to ignore assumed vector dependencies (proven dependencies area not ignored)

`#pragma vector always`

- Instructs the compiler to override any efficiency heuristic during the decision to vectorize or not

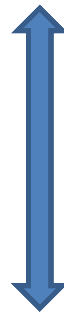
Vectorization: OpenMP 4.0 *simd*

- Intel took leadership in defining OpenMP 4.0 SIMD extensions
  - Several tuning options available

- Applied to a loop:
- `#pragma omp simd`

Ease of use

Thread Level Parallelism	**SIMD parallelism**
Auto Parallel	Auto vectorization
OpenMP threading	OpenMP 4.0 simd
Posix threads	Vectorization intrinsics



Programmer control

- Applied to a function to enable the creation of a version that can process arguments using SIMD instructions from a single invocation from a SIMD loop:
- `#pragma omp declared simd`

Vectorization: Intel Xeon Phi intrinsics

- IMCI intrinsics
  - The coding become hard
  - And the code is no more portable to common CPUs

```
for(i=0; i<N; i++)
     A[i] = A[i] + B[i];
```

```
for(i=0; i<N; i+=16) {
    __mm512 Avec = mm512load_ps(A+i);
    __mm512 Bvec = mm512load_ps(B+i);
    Avec = mm512add_ps(Avec, Bvec);
    _mm512_store_ps(A+i,Avec);
}
```

- The arrays float A[N] and float B[N] are aligned on a 64-byte boundary
- Variables Avec and Bvec are 512=16 x sizeof(float) bits

Exploiting cores

- MPI and OpenMP are the most common choices
  - Up to 60 MPI processes are reasonable for a single MIC
  - And 1 MPI process per MIC may be an interesting choice
  - The optimal choice between MPI and OpenMP depends on the application
- MPI Programming models, basically three configurations
  - Co-processor only (native mode)
  - MPI+Offload
  - Symmetric

Experimenting heterogeneity

- MPI communications are heterogeneous. Performances strongly vary!
- From some tests on the Eurora cluster at Cineca

	PingPong	SendRecv
CPU-CPU same node	5-11	5-22
CPU-CPU diff node	2.9	5
MIC-MIC same node	0.9	1.8
MIC-MIC diff node	0.9	1.6
CPU-MIC same node	5.9	11
CPU-MIC diff node	1.45	1.65

Symmetric mode: load balancing

- When running in symmetric mode, load balancing is a critical issue
  - Usual MPI decompositions assume homogeneous computing units
- Mixing MPI and OpenMP may help
  - Assign a different number of MPI processes to host and coprocessor
  - Exploit the full machine potential by means of OpenMP threads
  - E.g.
    - Host: 4 MPI ranks + 4 OpenMP threads
    - MIC: 8 MPI ranks + 30 OpenMP threads

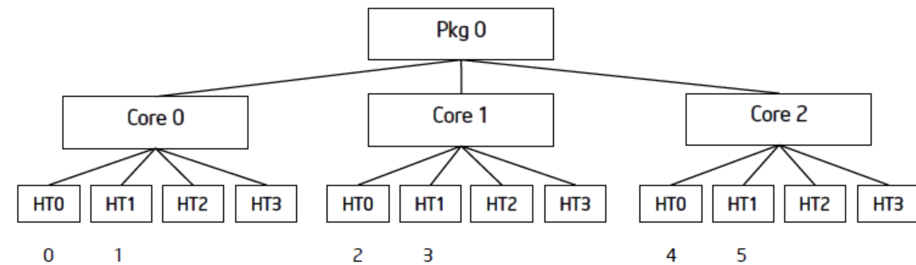
Threading models

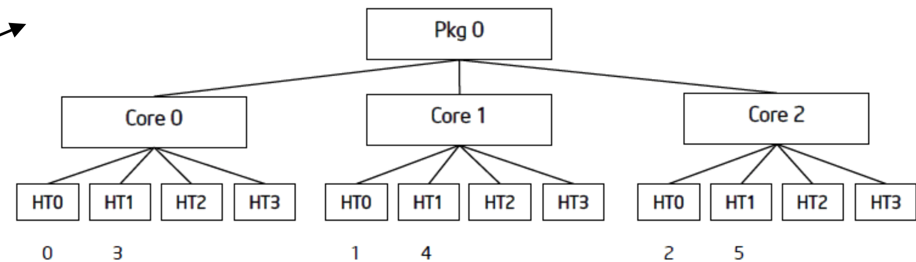
- Several threading models available
  - OpenMP
  - Fortran (2008) DO concurrent
  - Intel Cilk Plus
  - Intel Threading Building Block
  - Intel Math Kernel Library

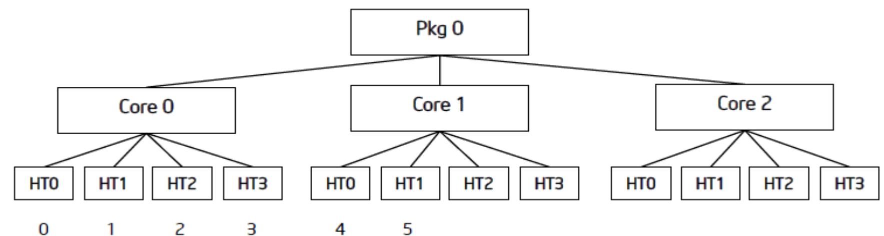
- OpenMP has clear advantages wrt portability
- In offload mode, it is possible (required) to tune both the host and coprocessor parameters (e.g. number of threads)

Thread Affinity

- Placement of threads on MIC cores and hardware threads
  - The basic configuration is
  - controlled by the variable
  - KMP_AFFINITY
  - Additional advanced
  - settings are possible too
  - Scatter
  - Balanced
  - Compact

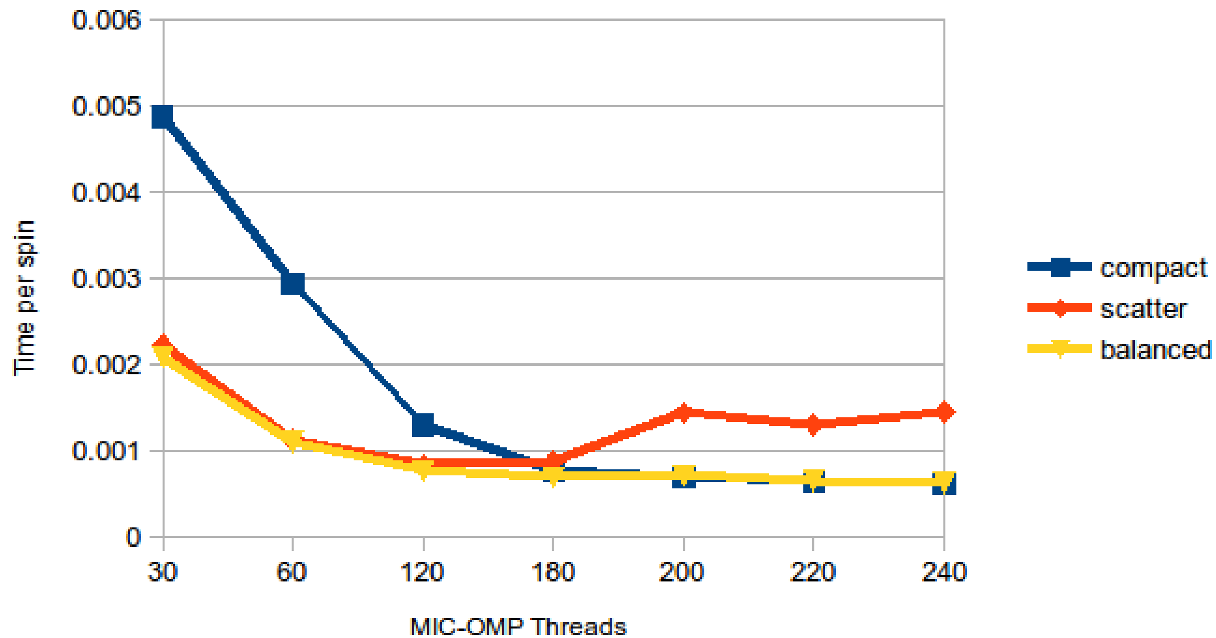






Affinity and performances

- The impact of affinity on performance may be very significant
- From a realworld example (3d-stencil code)



Collapse loops

- As recalled, the number of threads for each MPI process may become large (up to 240)
    - From different tests, it turns out that collapsing OpenMP loops results in improved performances
- From a realworld example (3d reacting Navier-Stokes equations)

MIC OMP threads	no-collapse	collapse
1	108.7	109.26
16	7.67	7.52
30	5.24	4.51
60	3.08	2.51
120	2.60	1.87
180	1.89	1.77
240	2.20	1.67

Tiling

- “Dividing a loop into a set of parallel tasks of a suitable granularity. In general, tiling consists of applying multiple steps on a small part of a problem instead of running each step on the whole problem one after the other. The purpose of tiling is to increase reuse of data in caches”

```
#pragma omp for collapse(2)
for (int z = 0; z < nz; z++) {
   for (int y = 0; y < ny; y++) {
      for (int x = 0; x < nx; x++) {

#define YBF 16
#pragma omp for collapse(2)
for (int yy = 0; yy < ny; yy += YBF) {
   for (int z = 0; z < nz; z++) {
      int ymax = yy + YBF;
      if (ymax >= ny) ymax = ny;
      for (int y = yy; y < ymax; y++) {
```

TLB cache thrashing

- "Depending on the memory patterns, possible TLB cache thrashing must be considered with care
  - Padding between allocated arrays may be a good solution
  - The problem may be difficult to analyze for non-HPC experts
- From a spin glass simulation code, the spin updating time has been measured against the padding pages between arrays

Padding pages	Time per spin
0	1.458
1	0.737
4	0.764
8	1.222
16	1.537
32	1.543

Intel VTune

- When getting unexpected performance results or whenever there is the need to have a deep understanding of the measured times, using Intel Vtune profiler is a good idea
- From the previous TLB thrashing example

Parameter	Non-padded	Padded
CPU Time	33459.268	31783.926
Clockticks	35199150000000.000	33436690000000.000
CPU_CLK_UNHALTED	35199150000000.000	33436690000000.000
Instructions Retired	724220000000	417970000000
CPI Rate	48.603	79.998
Cache Usage		
L1 Misses	13680450000	19016200000
L1 Hit Ratio	0.954	0.900
Estimated Latency Impact	2516.588	1732.644
Vectorization Usage		
Vectorization Intensity	10.913	10.248
L1 Compute to Data Access Ratio	3.138	4.783
L2 Compute to Data Access Ratio	68.417	47.795
TLB Usage		
L1 TLB Miss Ratio	0.033	0.064
L2 TLB Miss Ratio	0.026	0.000
L1 TLB Misses per L2 TLB Miss	1.252	1052.121
Hardware Event Count		
L2_DATA_READ_MISS_CACHE_FILL	464800000	548100000
L2_DATA_WRITE_MISS_CACHE_FILL	361900000	357000000
L2_DATA_READ_MISS_MEM_FILL	7220500000	7918400000
L2_DATA_WRITE_MISS_MEM_FILL	176400000	180600000

Data alignment/1

- DA is a method to force the compiler to create data objects in memory on specific byte boundaries. This is done to increase efficiency of data loads and stores to and from the processor.
  - For MIC memory movement is optimal when the data starting address lies on 64 byte boundaries
- Two steps are needed
- (1) Align the data

```
float A[1000] __attribute__((aligned(64)));
buf = (char*) _mm_malloc(bufsizes[i], 64);
real, allocatable :: a(:)
!dir$ attributes align:64 :: a
```

Data alignment/2

- (2) Use pragma/directives and clauses to tell the compiler that the accesses are aligned
  - For an i-loop that has a memory access of the form a[i+n1], the loop has to be structured in such a way that the starting-indices have good alignment properties.

```
__assume_aligned(a, 64);
__assume(n1%16==0);
__assume(n2%16==0);
for(i=0;i<n;i++) {
 // Compiler vectorizes loop with all aligned accesses
  X[i] += a[i] + a[i+n1] + a[i-n1]+ a[i+n2] + a[i-n2];
}
```

Streaming store and prefetch

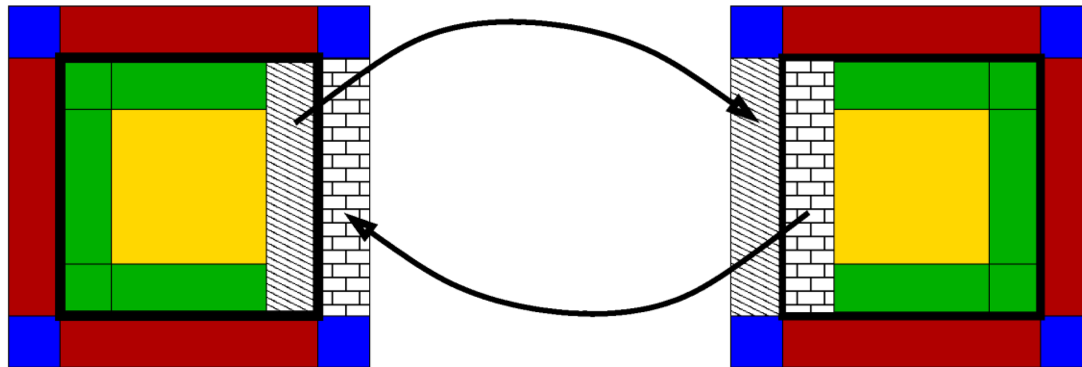
- Starting with Composer XE 2013 Update 1 compiler, streaming stores instructions are generated under certain conditions
  - Instructions intended to speed up the performance in the case of vector-aligned unmasked stores in streaming kernels where we want to avoid wasting memory bandwidth by being forced to read the original content of an entire cache line from memory when we overwrite their whole content completely
- Heuristics may be not sufficient: user can provide hints to the compiler, e.g.
- `#pragma vector nontemporal A`
- where A[i]=... is the store inside the loop

Native *vs* Offload

- Why offload mode?
- Cons
  - The porting is much more complex than to native mode
  - And the programmer must take care of host-coprocessors data exchanges which may be disastrous wrt performances
  - The symmetric mode allows to use both host and MIC at the same time
- Pros
  - it is also reasonable to assume that, the host being in charge of MPI calls (as it happens in offload mode), the MIC is free to execute, at its best, the computing intensive part of the code without wasting time in managing the communications

MPI optimizations: FDTD

- Consider a finite difference time domain code parallelized by standard domain decomposition. At each step:
  - (a) update boundary and bulk values
  - (b) exchange ghosts with neighboring processes



MPI optimizations: FDTD/2

- MPI patterns allow to overlap computations with communications (hiding the communication cost)
- Standard CPU pattern using MPI non blocking functions (available for MIC native mode as well)
  - Update boundary
  - Exchange ghost – MPI non blocking
  - Update bulk
  - Wait exchanges – MPI wait
- To achieve full overlapping, the bulk updating time must be larger than the communication time
- Using MIC (native), sometimes the final performances are far from optimal

MPI optimizations: FDTD/3

- MIC-Offload pattern (similar to multi-GPU approach)
  - Update boundary
  - Update bulk – asynchronous (non blocking)
  - Exchange ghost – MPI blocking
  - Wait bulk update

```
#pragma offload target(mic:0) … async(a)
{
<code to be offloaded>
}
CPU operations (e.g. MPI calls)
#pragma offload_wait(a)
```

MPI optimizations: HSG

- Scaling results from Heisenberg Spin Glass code
- Strong scaling for native/offload and sync/async versions

#MICS	Native-Sync	Native-Async	Offload-Sync	Offload-Async
1	0.709	0.717	1.049	1.078
2	0.484	0.431	0.558	0.527
4	0.445	0.325	0.335	0.281
8	0.376	0.246	0.219	0.167
16	0.343	0.197	0.154	0.113

- Weak scaling comparison with other architectures

#Procs	Size	CPU	GPU	MIC-n	MIC-o
1	256	3.73	0.67	0.78	1.34
8	512	0.48	0.068	0.25	0.17
Efficiency		96.2%	123%	39.9%	100%